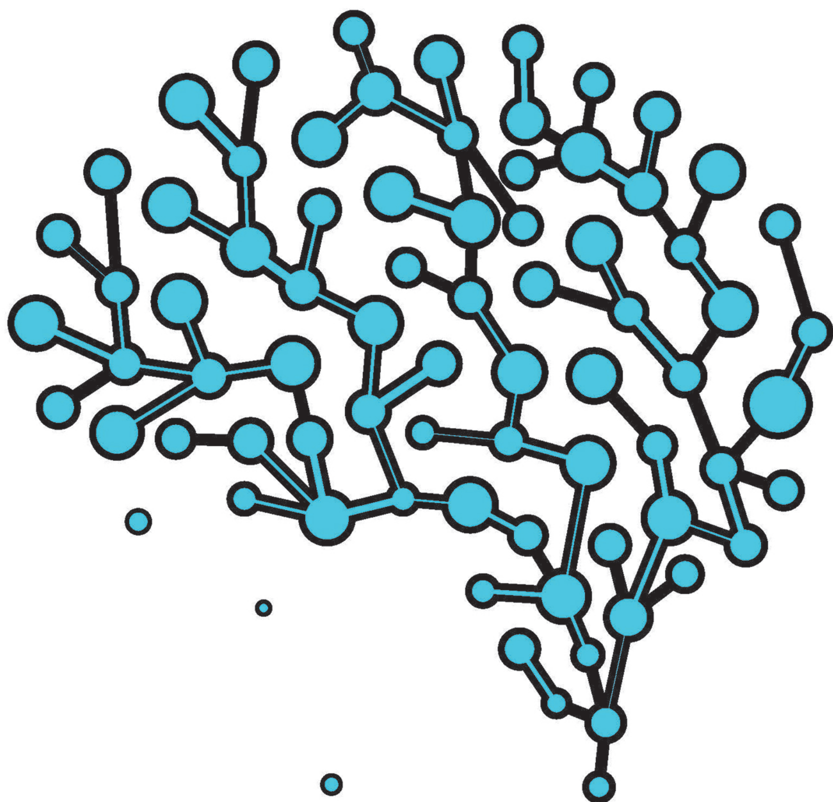
TURING 图灵原创



卷积神经网络的Python实现

单建华◎ 著

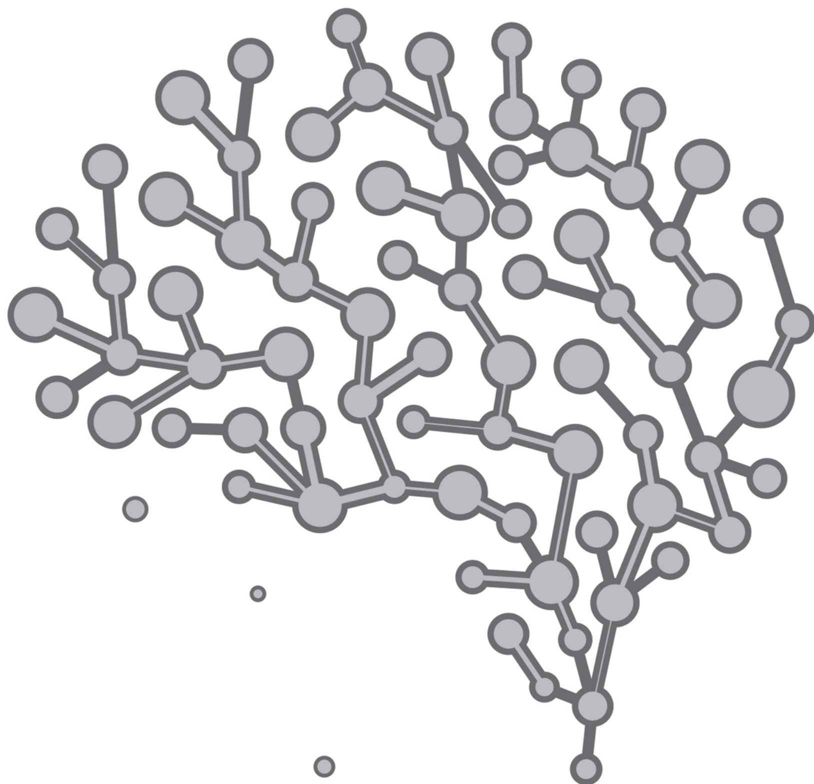
 中国工信出版集团  人民邮电出版社 POSTS & TELECOM PRESS

单建华

安徽工业大学教授、研究生导师，1998年进入中国科学技术大学就读，本硕博连读。博士毕业后从事图像处理和机器人研究。近几年研究深度学习，特别是卷积神经网络及其在汽车主动安全技术方面的应用，发表多篇论文并获得多个专利授权。

数字版权声明

TURING 图灵原创



卷积神经网络的Python实现

单建华◎著

人民邮电出版社
北京

图书在版编目（C I P）数据

卷积神经网络的Python实现 / 单建华著. -- 北京 :
人民邮电出版社, 2019.1
（图灵原创）
ISBN 978-7-115-49756-7

Ⅰ. ①卷… Ⅱ. ①单… Ⅲ. ①人工神经网络－软件工具－程序设计 Ⅳ. ①TP183

中国版本图书馆CIP数据核字(2018)第239530号

内 容 提 要

卷积神经网络是深度学习最重要的模型之一。本书是卷积神经网络领域的入门读物，假定读者不具备任何机器学习知识。书中尽可能少地使用数学知识，从机器学习的概念讲起，以卷积神经网络的最新发展结束。

本书首先简单介绍了机器学习的基本概念，详细讲解了线性模型、神经网络和卷积神经网络模型，然后介绍了基于梯度下降法的优化方法和梯度反向传播算法，接着介绍了训练网络前的准备工作、神经网络实战、卷积神经网络的应用及其发展。针对每个关键知识点，书中给出了基于 NumPy 的代码实现，以及完整的神经网络和卷积神经网络代码实现，方便读者训练网络和查阅代码。

本书既可以作为卷积神经网络的教材，也可以供对卷积神经网络感兴趣的工程技术人员和科研人员参考。

◆ 著　　　　单建华
责任编辑　王军花
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　印刷

◆ 开本：880×1230　1/32
印张：7.5　　　　彩插：2
字数：166千字　　　　2019年1月第1版
印数：1 – 3 000册　　　　2019年1月北京第1次印刷

定价：49.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广登字 20170147 号

前　　言

近几年，深度学习在学术界和工业界掀起革命，Python 语言作为机器学习的首选语言也异军突起。然而，大多数 Python 语言的教材侧重语法，很少有项目实战，深度学习方面的教材侧重理论，特别是针对卷积神经网络，很少涉及源码解读和实现。因此我试图通过本书，使读者掌握 Python 语言并深入理解卷积神经网络，并能用 Python 语言实现卷积神经网络结构。

为了使尽可能多的读者对卷积神经网络有所了解，本书假定读者不具备任何机器学习知识。同时，我尽量少地使用数学知识，只要掌握向量、矩阵及矩阵乘法的定义，就能理解本书中除了优化之外的所有章节；优化问题涉及的数学知识有导数、偏导数、泰勒展开式和链式法则。学习 Python 语言需要掌握一些基本知识，如 list 结构、`for` 循环、函数、类和 NumPy 模块。因此，本书适合具有一定 Python 基础的大学二年级以上的理工科本科生和研究生，具有类似知识基础的工程技术人员和科研人员，以及对卷积神经网络有一定了解，想进一步理解的人士阅读。

全书共 10 章，分为 3 个部分：第一部分（第 1 章 ~ 第 4 章）介绍了机器学习的基本概念、线性模型、神经网络和卷积神经网络模型，采用一条主线贯穿这 3 种模型，层层递进，降低了学习难度；第二部分（第 5 章和第 6 章）介绍了基于梯度下降法的优化方法和梯度反向传播算法，详细地给出了各种模块的反向传播代码，读者掌握了这些方法后，如果遇到新的模块，就能独立实现其反向传播代码；第三部分（第 7 章 ~ 第 10 章）介绍了训练网络前的准备工作、神经网络实战、卷积神经网络的应用及其发展，特别详细地给出了数据归一化、梯度检测、批量归一化（BN）及各种经典的卷积神经网络结构，如 VGGNet、GoogLeNet、ResNet、SENet 和轻量网络 MobileNetV2。除第 1 章外，每个关键知识点都给出了基于 NumPy 的代码实现，特别是第 8 章和第 9 章这两章，给出了完整的神经网络和卷积神经网络代码实现，方便读者训练网络和查阅代码。

如果你只是想了解卷积神经网络的原理，可只看第一部分和第 10 章。如果你只是想更好地利用 TensorFlow 等框架来训练网络结构，可不用看第 6 章，通过学习第 5 章就能正确选择合适的优化算法。如果你想不借助 TensorFlow 等框架实现各种网络结构，则需学习所有章节，特别是第二部分。

本书中的代码[①]的一个突出优点是接近伪代码，几乎不言自明，十分容易读懂。如果读者没有 Python 基础，不看代码也能掌握卷积神经网络的知识。当然，强烈建议读者手工输入代码，泛读或精读代码，

① 本书代码可以从图灵社区（iTuring.cn）本书主页免费注册下载。

注意其实现细节。这能加深对卷积神经网络的理解，同时也是一个极好的 Python 语言项目实战训练。

卷积神经网络发展极为迅速，各种新网络结构层出不穷，理论也在一步一步发展，本书只是对卷积神经网络最基本、最核心的概念，以及最重要的网络结构进行介绍。我自认才疏学浅，略知皮毛，更兼精力有限，书中错谬之处在所难免，若蒙读者不吝告知，将不胜感激。

我的邮箱是 shanjianhua.vip@qq.com，如有疑问，欢迎垂询。

最后要感谢我的家人，特别感谢我的爱人佘慧莉女士，感谢她一直以来对我的理解和支持。当然，也要感谢使本书顺利出版的编辑王军花和武芮欣两位女士。

单建华

2018 年 5 月于马鞍山

目　　录

第一部分　模型篇

第二部分 优化篇

第三部分　实战篇

第一部分

模型篇

第 1 章

机器学习简介

1.1　引言

我们以生活中常见的挑选西瓜为例。到了夏天，大家喜欢吃西瓜，希望买到好西瓜。怎么挑到好西瓜呢？我们会根据西瓜的一些属性特点（如根蒂、敲声、触感和纹理等）进行挑选，一般认为根蒂凹陷、敲声浑响、触感硬滑和纹理清晰的西瓜是好瓜。这些挑选西瓜的经验是人类掌握的知识，是在无数次挑选西瓜后总结出来的。

具体怎么掌握挑选西瓜的知识呢？假设开始时，我们对西瓜没有任何了解，西瓜的好坏只能随机猜测。为了提高判断的准确率，一般来说会这么做：拿到一个西瓜，切开，亲口品尝，确定西瓜的好坏，然后记录这个西瓜相关的属性特点。这时由于掌握的知识十分有限，所以必须记录大量的属性，以确保没有遗漏重要的相关属性。随着品尝西瓜数量的增加，逐渐能总结、归纳出一些挑选西瓜的知识。总结出知识的可靠性也随着西瓜数量的增加而提高，但是提高速度会越来

越慢，最后有可能趋于饱和，即品尝再多的西瓜，也不能产生新的属性特点。

机器学习的目的就是让计算机像人类一样，能区分西瓜的好坏。那么如何让计算机学习这些知识，就是机器学习的核心内容。

1.2　基本术语

机器学习的过程与人类学习的过程十分类似，必须有大量的西瓜数据，这些数据构成训练数据集。机器从这些训练数据集中学习，这个过程类似人类的归纳推理，从而获得隐藏在数据背后的模型，然后利用这个模型对不在训练集中的新西瓜进行预测，根据预测的准确率来评判模型的优劣。一般来说，训练数据集越大，机器学习到的模型的预测性能就越好。

机器学习的本质是计算机通过数据来拟合隐藏在数据背后的模型。注意是拟合模型，不是推导模型。物理学家的研究目的是推导出确定模型，如建立自由落体公式，机器学习专家研究的目的是用数据来拟合模型，如根据自由落体不同时刻下落距离的一系列数据，建立拟合模型来预测任意时刻的下落距离。所以如果一个问题已经存在确定模型，就不需再用机器学习。

怎么确定机器真正学习到了挑选西瓜的知识呢？这只能提供一些不在训练集中出现过的新西瓜，让机器根据它们的根蒂、敲声、触感和纹理等属性来判断好坏，判断的准确率越高，就说明机器掌握挑西

瓜的知识越好。

我们把每个西瓜数据称为一个样本，每个样本数据包括两部分：一部分是西瓜属性数据，也称为特征属性，如根蒂、敲声、触感和纹理等；另一部分是西瓜好坏的标签，表示为(x, y)，其中 x 是属性数据取值，如根蒂 = 蜷曲、敲声 = 清脆、触感 = 硬滑、纹理 = 模糊等，y 是标签，表示西瓜的好坏。标签取值为离散值，是**分类问题**：如果离散值只取两个，则是二分类问题，如本例中西瓜标签取“好”和“坏”两个值；如果离散值取多个，则是多分类问题，如 0 到 9 的数字识别是 10 分类问题，而二分类问题是多分类问题的基础。标签取值为连续值，是**回归问题**。如果不仅要判断西瓜好坏，还要进一步判断西瓜好坏的程度（0 表示最坏，1 表示最好，0.35 表示较坏，0.75 表示较好等），就是回归问题。

机器学习得到的模型，本质上是得到从特征属性 x 到标签 y 的**映射** f：$y = f(x, w)$，其中 w 是模型参数。在神经网络模型中，w 是所有权重参数，有的映射不包含参数 w，如最近邻和朴素贝叶斯。对于二分类问题，通常令 $y = \{+1, -1\}$；对于多分类问题，则 $|y| > 2$；对于回归问题，$y \in \mathbf{R}$，其中 $\mathbf{R}$ 是实数集。本书中映射和模型两个词同义，有时混用，请读者注意。

如何评价模型好坏呢？可以让模型预测新样本，得到预测标签 $\hat{y}$，$\hat{y} = f(x, w)$。本书中，如果标签上有帽子上标，则表示**预测标签**，无上标则表示**真实标签**。预测标签与真实标签进行比较，以评判预测效果，进而评价模型的好坏。注意评价模型好坏时，必须使用新样本，

即没有在训练集中出现过的样本，这样才能真实评价模型的性能，模型预测新样本的能力称为**泛化性能**。评价一个模型泛化性能时采用的样本集称为**测试集**。

如果采用训练样本进行预测，则机器学习算法可以采用偷懒的办法来达到极高的准确率，甚至 100%。因为算法只要死记硬背所有的训练样本，预测时使用查询方法，准确率即可达到 100%，但实际上算法没有进行任何有意义的学习。这和学生学习类似，老师为了评估学生的学习效果，需要组织考试，如果把答案提前告诉学生，则学生只需死记硬背，就能考 100 分，但这完全不能真实反映学生的学习能力，只有考试题目学生从来没有做过，才能较为真实地评估学生学习的效果。那是不是只要试卷上的题目是学生没有做过的，就是好试卷？显然不是，题目必须是专家精心设计的，能涵盖大部分知识点、难易适中。测试集和训练集好比试卷，必须精心设计，尽量覆盖样本分布空间。

1.3　重要概念

为了使读者深入理解机器学习，这里需要补充说明几个重要概念。读者应该反复研读，并结合后面的知识与实际项目进行理解。

1. 相关属性和无关属性

西瓜的属性有很多，我们为什么选择根蒂、敲声、触感和纹理进行判断，不选大小、外形等属性进行判断呢？这是因为根蒂、敲声、

触感和纹理属性与西瓜好坏密切相关，是**相关属性**，大小、外形等属性与西瓜好坏无关，是**无关属性**。无关属性对于判断西瓜好坏不能提供任何有价值的信息，所以不需要这些无关属性。

那么，怎么知道属性是相关属性还是无关属性呢？这是在无数次挑选西瓜时总结出来的知识，属于领域知识。在一个具体的机器学习任务中，必须充分利用领域知识，挑选出相关属性，剔除无关属性。如果没有领域知识，则只能采用大量属性，这会极大地增加机器学习的难度并且降低机器学习的效果。

2. 有效学习

模型对新样本的泛化性能必须大于随机猜测，才能说明模型进行了**有效学习**。对于二分类问题，如果准确率是 50%，则说明机器没有进行任何学习，只有大于 50%，才能说明机器进行了学习。一般来说，准确率不可能达到 100%，只能接近，因为现实世界中的样本无穷无尽，特征取值也各不相同，很难完全掌握，总会存在误判。

大家可能会设想，随着掌握的领域知识不断增长，总有一天能完全准确判断西瓜好坏，准确率达到 100%，完美地建立起西瓜好坏的模型。如果真有这么一天，判断西瓜好坏就不应该是机器学习研究的领域，因为我们已经有了判断西瓜好坏的确定模型，直接根据模型就能判断，没有必要采用机器学习了。是否存在一个确定模型能准确判断西瓜好坏，这是一个终极难题，既不能肯定，也不能否定。

3. 从数据中学习

机器仅能从数据中获得分类的能力，根据指令一步一步地对样本进行分类，人类不能显式编程告诉它。举个数据排序的例子，人类开发了很多排序算法，如冒泡、快排等，如果把这些排序算法进行显式编程，计算机获得了排序能力，而且准确率 100%，但这不是机器学习。如果只是提供给计算机很多无序和有序的数据对，计算机仅根据这些数据对进行学习，获得了排序能力，则是机器学习，但准确率一般达不到 100%。学习到的排序知识存储在模型参数 w 中。

4. 映射

对于不同的机器学习方法，映射 f 的函数形式是不同的，如线性模型、SVM 和神经网络的映射 f 是不同的。在同一机器学习方法中，映射 f 的函数形式是相同的，不同的只是参数 w。本书后面会详细介绍线性模型和神经网络模型，对于 SVM 模型，读者可查阅相关的资料。

5. 机器学习中的优化难题

如何获得最优参数 w，使模型 f 的泛化性能最好？这是数学中的参数优化问题。读者如果对高中数学和高等数学印象深刻的话，应该做过很多求函数极值的题目，机器学习中的参数优化问题和数学求极值题目没有本质差别，只是难度更大。根据连续函数的极值定理，连续函数的极值条件是偏导数为 0。以上知识点，读者如果不熟悉，强烈建议读者学习。机器学习中求最优参数完全是数学中的参数优化问题吗？如果这样认为，就错了，下面将通过 3 点来解释求最优参数不

仅仅是数学参数优化问题。

第一，模型 f 的泛化性能跟训练数据集密切相关，针对不同的训练集，其泛化性能的差别可能很大。因此，建立高质量的训练集是机器学习的重要任务，占整个工作量的 60% 以上。如何建立高质量的训练集呢？还是以西瓜任务为例。首先，训练集应该尽可能包含各种各样的西瓜（如品种、产地、栽培土壤和施肥浇水等），这就是所谓的见多识广。其次，应该尽可能多地提取与任务相关的属性。这样的训练集很容易训练出效果好的模型，即使使用的机器学习算法很普通。建立高质量的训练集需要大量的人力、物力和领域知识，一般的研究者很难建立高质量的训练集，而没有好的训练集，就难以开发好的机器学习算法。为了使广大研究者能专心进行机器学习算法研究，机器学习界有许多著名的公开数据集。读者学习机器学习算法时，最好使用这些数据集。最后强调，不同的学习算法必须在同一个测试集进行比较，才能区分优劣。

第二，模型 f 的函数形式虽然是统一规范的，但不同的机器学习方法采用不同的函数形式，如神经网络和支持向量机就不同，这导致学习效果也不同。如何选择好的函数形式，一直是机器学习领域最核心和最富创新性的课题。本书讲的卷积神经网络之所以在图像分类中获得空前胜利，就是因为卷积运算巧妙地把视觉神经领域知识融入到函数形式中。VGGNet、GoogLeNet 和 ResNet 等著名的卷积网络模型就是映射 f 采用的函数形式稍微不同，使这些模型的效果各不相同。如何设计更好的 f 将一直是机器学习研究的核心课题。

第三，评价模型的泛化性能时，必须使用测试集，但是训练过程中使用的是训练集，测试集是“不存在”的，这样可能存在一种情况：模型对训练集有很好的拟合效果，但对测试集拟合效果一般。这是机器学习特有的现象，称为过拟合。过拟合现象十分普遍，理论上讲，任何机器学习算法都有过拟合风险，只是强弱问题。过拟合表现为只要精心调整参数 w 或一直训练下去，模型在训练集上的性能就会越来越好，但测试集上的性能则不会，它存在拐点，拐点之后性能会逐渐变差。过拟合的通常解释是：因为模型的学习容量随着参数的精心调整和训练时间的增加而越来越大，训练集的规模却是固定的，导致模型把训练集学习得过分好，其中的一些噪声也学为样本的固有模式，因此泛化到新样本时就容易出错。过拟合是机器学习中一个难以克服的问题，因为其产生的本质原因在理论上还没有解释清楚，一般采用正则化来缓解过拟合。

上面这 3 点也是机器学习研究的核心内容，本书主要讲解基于梯度下降法的参数优化方法和神经网络模型，缓解过拟合也是重点之一。对于如何建立高质量的训练集，这和领域知识密切相关，本书暂不涉及，本书将采用公开数据集。

6. 过拟合实例

为了使读者对过拟合有感性认识，这里有一个十分经典的例子：多项式数据拟合。假设数据点 (x, y) 的真实模型是二次曲线，由于噪声，每个采样点都会偏离理想值。为了减小训练误差，极端情况下，为使训练误差为 0，即拟合曲线严格通过每个数据点，从理论上可证：

对于有 N 点的数据，则拟合曲线必须是 $N-1$ 次曲线，如一次函数通过两点，二次函数通过三点等。一般情况下，拟合曲线是高次函数，N 至少取 10 以上。高次函数拟合容易出现数值计算不稳定、曲线振荡等情况，对于真实模型是二次曲线的数据拟合任务，高次函数的泛化性能会远低于低次函数。这里所说的学习容量，就是拟合曲线的次数，其值越高，容量越大，越能拟合变化剧烈的数据。学习“过好”就是很好地拟合了训练数据，导致把噪声作为数据变化趋势，产生过拟合。怎么缓解过拟合呢？读者可能会说，用真实模型的二次曲线拟合。但在实际情况中很难通过数据变化趋势准确判断曲线次数，所以只能大概估计，采用 3 次或 4 次曲线进行拟合。为了缓解曲线振荡，对方程系数（即参数 w）进行约束，使系数的绝对值尽可能小，这就是正则化。但曲线次数又不能取得太低，这会使学习容量过低，难以拟合数据变化趋势。所以实际做法是：采用学习容量较大的模型，使学习过程相对容易，利用正则化缓解学习容量较大所引起的过拟合。

与过拟合相对的是欠拟合，欠拟合指模型的学习容量过低，不能学习到训练样本的一般性质。欠拟合很容易克服，只需增大模型容量即可。

7. 训练集、验证集和测试集

如何正确评估模型的泛化性能？由于过拟合，要正确评估模型的泛化性能并不是一件简单的事情，往往会过于乐观，高估了模型的泛化性能。以多项式数据拟合为例，对于固定次数的曲线，利用训练集能学习到最优系数 w。采用不同次数的曲线拟合时，其拟合效果

一般是不同的，怎么确定曲线最优次数呢？只能用“测试集”测试不同次数曲线的拟合效果，我们一般把这个“测试集”叫作验证集，用于选取拟合效果最优的曲线作为最终模型。如果认为模型的泛化性能就是验证集的最优拟合效果，就高估了模型的泛化性能。因为存在过拟合，所以验证集最优的模型不一定在其他测试集会最优，还需要另一个测试集评估该最优模型的泛化性能。因此，完整的机器学习需要 3 个数据集：训练集、验证集和测试集。训练集寻找最优参数 w，验证集决定最优曲线方程次数，测试集评估模型的泛化性能。

8. 超参数和参数

机器学习一般包含两类参数：超参数和参数。超参数的数目通常不多，在 10 以内；参数的数目可能很多，如卷积神经网络中有近千万个参数（权重）。曲线拟合中，方程的次数就是超参数，多项式的系数就是参数。这两种参数的调参方式不同，超参数取值一般是人工设定的，参数值是根据参数优化算法自动寻优的。超参数的取值对模型泛化性能有重大的影响，验证集就是用来决定最优超参数取值的。目前，出现了很多超参数自动优化算法。

9. 如何提高模型泛化性能

当模型的泛化性能不能满足要求时，可以从两个方面提高模型的泛化性能。其一是提高模型的学习容量，把训练误差降低到满意的程度，同时增大正则化强度，降低过拟合风险。其二是构造更好的数据集，这包括增大样本数量，调整样本以更加全面地覆盖样本分布空间，利用领域知识提取更好的特征。

10. 模型输出

模型的输出，如果是回归问题，则直接输出实数。在多分类任务中，输出一般是向量，向量的维数等于类别数，元素值是实数。元素值表示对应类别的得分，得分越大表示该样本与对应的类别越相似，故最大元素值对应的类别就是最终预测的类别。二分类是多分类的特例，只需输出一个实数值，并与 0 比较，大于 0 预测为正类，小于 0 预测为负类。

1.4 图像分类

人类每时每刻都在进行图像分类，对人类来说，图像分类“简单”到甚至察觉不到。走在街道上，你要判断路上的车辆、行人、红绿灯、道路和人行横道等。

图像分类是指：对于一幅给定的图像，模型需要判定它属于所给定的类别中的哪一个，如一幅图像属于集合{卡车，公交车，私家车，货车}中 4 类的概率分别是多少。这个任务对人类看似很简单，却一直是计算机视觉的一个核心和难点问题。图像分类应用很广泛，计算机视觉的很多问题（如物体检测、图像分割）的基础都是图像分类。

1. 计算机“看到”的图像是什么

计算机“看”不到图像的内容，对它而言，图像是巨大的数值矩

阵，矩阵元素表示像素的颜色信息。例如，某幅图像分辨率为 1280 × 720，表示图像有 1280 × 720 个像素点，则存储为 1280 × 720 的矩阵。对于彩色图像，每个像素点有红、绿、蓝（RGB）3 个颜色的通道值，每个值在 0（黑）到 255（白）之间；对于灰度图像，每个像素点有亮度 1 个通道值，每个值也在 0 到 255 之间。计算机只能根据"看到"的这个数值矩阵去判定图像内容，采用 $H \times W \times C$ 表示图像，其中 H 是图像高度，W 是宽度，C 是通道数。

2. 图像分类的挑战

图像分类对人类很简单，但对计算机而言，会面临诸多困难，主要有如下几点。

- **视角变化**：摄像机可以从多个角度拍摄同一个物体，不同角度的图像内容不同。
- **大小变化**：同一类物体有大有小，而且拍摄距离对大小的影响很大。
- **形变**：很多物体不是刚体，会产生形变。
- **遮挡**：目标物体可能会被挡住，有时候物体只有一小部分是可见的。
- **光照条件**：强光和暗光环境下拍摄，像素的亮度差别非常大。
- **背景干扰**：物体可能混入复杂背景中，难以辨认。
- **类内差异**：同类物体之间的外形差异很大，比如椅子。这类物体有许多不同的子类，每个子类都有自己独特的外形。

好的图像分类模型必须能克服上述变化及其组合，同时对类间差异足够敏感。

3. 语义鸿沟

我们知道，计算机只能“看到”一个个像素值堆积成的矩阵，并非人类看到的物体内容。人类在判别图像内容时，不是建立在图像像素值这种低层视觉特征上，而是建立在对图像语义理解的基础上，如人脸由眼睛和嘴巴等组成。这种语义理解无法从图像的低层视觉特征中直接获得，而是由人们长期积累的大量先验知识帮助判断。换言之，人们是依据图像的语义信息来进行图像判别的，这产生了人所理解的“语义相似”与计算机理解的“视觉相似”之间的“语义鸿沟”。语义鸿沟就是由于计算机获取的图像信息与人类对图像理解的语义信息的不一致性而导致的低层信息和高层信息之间的距离。像素信息不包含任何语义信息，计算机难以从像素中提取语义信息，这是图像分类难度大的内在原因。

4. 数据驱动方法

那么，如何写一个图像分类的算法呢？这和排序算法大不一样，怎么写一个从图像中认出车的算法？因此，与其在代码中直接写明各类物体到底“看起来”是什么样，不如采取小孩看图识物的方法：给计算机很多物体图像，然后采用机器学习方法，让计算机自己学习区分每个类别物体图像的视觉表现特征。这种方法，就是数据驱动方法，也就是机器学习方法。不要直接告诉计算机该“怎么做”，而是给计算机大量的实例，让计算机自己学会“怎么做”。

1.5　MNIST 数据集简介

给计算机看的大量实例构成数据集。MNIST 数据集是手写数字集，在图像分类中十分著名。深度学习三巨头之一 Geoffrey Hinton 称其为机器学习界的小白鼠，表明其基础和简单的特点。三巨头中的 Yann LeCun，也是卷积神经网络之父，他在该数据集上首次实现了卷积神经网络，但由于该数据集很简单，不能充分发挥卷积神经网络的优势，被当时的支持向量机（SVM）盖过了风头。直到 ImageNet 数据库出现，才使卷积神经网络再度火起来。由此可见，MNIST 作为入门数据集，是非常合适的。

MNIST 来自美国国家标准与技术研究院，训练集由 250 个不同的人手写的数字构成，其中 50% 是高中学生，50% 来自人口普查局的工作人员；测试集也是同样比例的手写数字。MNIST 数据集可在 http://yann.lecun.com/exdb/mnist/上进行下载，训练集包含 60 000 个样本，测试集包含 10 000 个样本。MNIST 数据集中的每张图像都是灰度图像，由 28×28 个像素点构成，每个像素点用一个灰度值表示，如图 1.1 和图 1.2 所示。每张图像的标签就是手写数字的类别标签（整数 0~9）。



图 1.1　MNIST 数据集 0 ~ 9 数字样本

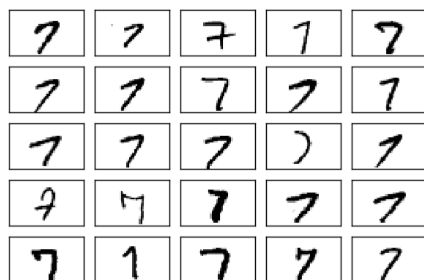


图 1.2　MNIST 数据集数字 7 样本

其他知名数据库有 CIFAR-10、CIFAR-100 和 ImageNet 等，特别是 ImageNet，推动了深度学习的发展。

第 2 章

线性分类器

2.1 线性模型

第 1 章介绍了机器学习模型，它的本质上是得到由属性 x 到标签 y 的映射 f：$y=f(x,w)$（其中 w 是模型参数），通过最优化理论获得最优值。多分类任务中，输出 $\boldsymbol{y}$ 一般是向量，称为输出向量。输出向量的维数等于类别数，其元素值是实数，表示对应类别的分值，分值越大，该样本与对应的类别就越相似，故最大元素值对应的类别就是最终预测的类别。当映射 $\boldsymbol{y}=f(\boldsymbol{x},\boldsymbol{w})$ 采用最简单的线性函数时，就得到线性分类器。线性函数虽然简单，却是理解神经网络和卷积神经网络的基础，希望读者重视。

假设训练集 $\boldsymbol{x}_i\in\mathbf{R}^D$，每个样本对应的类别标签为 $\boldsymbol{y}_i$，$i=1,2,\cdots,N$，$\boldsymbol{y}_i\in 1,\cdots,K$，这表示训练集有 N 个样本，每个样本的维度是 D（即有 D 个属性），共有 K 个类别。

例如，在 MNIST 中，有 $N=60\,000$ 个训练样本，每个图像有 $D=$

$28\times28\times1=784$ 个维度，$K=10$，这是因为有 10 个数字。图像的每个像素是一个特征属性。线性模型和神经网络模型中，图像矩阵需要拉伸为一个向量，卷积网络不需要。

2.1.1 线性分类器

属性 $\boldsymbol{x}_i$ 和参数 $\boldsymbol{w}$ 的线性函数为：

$$s=f(\boldsymbol{x},\boldsymbol{w},\boldsymbol{b})=\boldsymbol{x}_{i1}\boldsymbol{w}_1+\boldsymbol{x}_{i2}\boldsymbol{w}_2+\cdots+\boldsymbol{x}_{iD}\boldsymbol{w}_D+\boldsymbol{b} \tag{2.1}$$

其中输出 s 是分值，D 个 $\boldsymbol{w}_i$ 组成参数 $\boldsymbol{w}$，也称权重向量，$\boldsymbol{b}$ 是偏置参数。由于 s 是标量，只能表示某个类别的分值，不能表示所有类别的分值。为了得到 K 个分值，则需 K 个线性函数：

$$\begin{aligned}
\boldsymbol{s}_1&=\boldsymbol{x}_{i1}\boldsymbol{w}_{11}+\boldsymbol{x}_{i2}\boldsymbol{w}_{12}+\cdots+\boldsymbol{x}_{iD}\boldsymbol{w}_{1D}+\boldsymbol{b}_1\\
\boldsymbol{s}_2&=\boldsymbol{x}_{i1}\boldsymbol{w}_{21}+\boldsymbol{x}_{i2}\boldsymbol{w}_{22}+\cdots+\boldsymbol{x}_{iD}\boldsymbol{w}_{2D}+\boldsymbol{b}_2\\
&\vdots\\
\boldsymbol{s}_K&=\boldsymbol{x}_{i1}+\boldsymbol{w}_{K1}+\boldsymbol{x}_{i2}\boldsymbol{w}_{K2}+\cdots+\boldsymbol{x}_{iD}\boldsymbol{w}_{KD}+\boldsymbol{b}_K
\end{aligned} \tag{2.2}$$

第 i 个方程表示第 i 类的分值函数，也称为第 i 个分类器。为了简化书写，可以写成矩阵形式。（刚开始可能不习惯矩阵形式，但一定要逐渐习惯。）分两步改写比较容易理解，第一步：

$$\begin{aligned}
\boldsymbol{s}_1&=\boldsymbol{x}_i\boldsymbol{w}_1+\boldsymbol{b}_1\\
\boldsymbol{s}_2&=\boldsymbol{x}_i\boldsymbol{w}_2+\boldsymbol{b}_2\\
&\vdots\\
\boldsymbol{s}_K&=\boldsymbol{x}_i\boldsymbol{w}_K+\boldsymbol{b}_K
\end{aligned} \tag{2.3}$$

注意此时各个 $\boldsymbol{w}_i$ 是列向量，$\boldsymbol{x}_i$ 是行向量，$\boldsymbol{x}_i\boldsymbol{w}_i$ 是向量内积。第二步：

$$s = x_i W + b \tag{2.4}$$

注意此时 $\boldsymbol{s}$ 是分值行向量，$\boldsymbol{b}$ 是偏置行向量，$\boldsymbol{W}$ 是权重矩阵，每列由 w_i 个列向量构成。

在 MNIST 中，各向量或矩阵的维度为：$\boldsymbol{s}$ 是$[1 \times K] = [1 \times 10]$，$\boldsymbol{b}$ 是$[1 \times K] = [1 \times 10]$，$\boldsymbol{x}_i$ 是$[1 \times D] = [1 \times 784]$，$\boldsymbol{W}$ 是 $[D \times K] = [784 \times 10]$。

如果把上面矩阵形式的线性模型中的每个向量或矩阵当作标量，则线性模型就变成一元一次方程，变成最简单的线性模型了。

对于线性模型，需要强调如下几点。

第一，进行多分类，只需一个矩阵乘法 $\boldsymbol{x}_i\boldsymbol{W}$ 和矩阵加法，每个分类器就是 $\boldsymbol{W}$ 的一个列向量。

第二，训练集$(\boldsymbol{x}_i, \boldsymbol{y}_i)$是给定且不可改变的，可变的是权重 $\boldsymbol{W}$ 和偏置向量 $\boldsymbol{b}$。机器学习的目的就是通过优化算法得到这些参数的最优值，使得模型计算出来的分值向量和训练集中样本的真实类别标签相符。何谓相符，详见 2.2 节。

第三，模型训练的结果是得到最优参数 $\boldsymbol{W}$ 和 $\boldsymbol{b}$。一旦训练完成，就不再需要训练集。因为预测样本类别时，只需计算分值向量，不需要训练集。

2.1.2　理解线性分类器

线性分类器计算图像的所有像素值与权重的内积，从而得到该分类器的类别分值。根据权重符号，分类器对图像中的像素表现出喜爱

（符号为正）或者厌恶（符号为负）。权重符号为正，乘以像素值（为正），能提高分值；权重符号为负时，则减小分值。如果图像属于某类，则分值要大；不属于某类，则分值要小。所以，可以想象“飞机”图像被大量的蓝色（对应“天空”）所包围，那么“飞机”分类器在蓝色通道上有很多的正权重（提高“飞机”的分值），而在绿色和红色通道上的权重为负的就比较多（降低“非飞机”的分值）。

将图像看作高维空间的矢量：我们把图像拉伸成为高维空间的一个列向量，就可以将其看作这个空间中的一个矢量（矢量的起始点是坐标轴原点，MNIST 中的每张图像是 784 维空间中的一个矢量）。整个数据集就是矢量集，每个矢量都带有一个类别标签。每个样本的分值是矢量 $\boldsymbol{x}_i$ 和权重列向量 $\boldsymbol{w}_i$ 的内积，$\boldsymbol{w}_i$ 可以看作高维空间中的矢量，内积可以看作矢量 $\boldsymbol{x}_i$ 向矢量 $\boldsymbol{w}_i$ 的投影长度。如果把内积看作物理中两个矢量的点积，高维空间压缩为二维空间，则能可视化分类器。举个例子，把 $\boldsymbol{w}_i$ 看作二维平面的力，$\boldsymbol{x}_i$ 看作速度，则内积 $\boldsymbol{x}_i^{\mathrm{T}}\boldsymbol{w}_i$ 就是功率。分类器就是寻找最优力矢量，使速度 $\boldsymbol{x}_i$ 与对应类别的力 $\boldsymbol{w}_i$ 的功率最大。

线性分类器看作模板匹配：$\boldsymbol{w}_i$ 对应一个类别模板，分类就是找到 $\boldsymbol{x}_i$ 和哪个模板最相似。从这个角度来看，线性分类器就是利用学习得到的模板，做模板匹配，使用负内积来计算图像与模板的距离，距离越小，说明越相似。

线性分类器只能对线性可分的样本进行有效分类，线性可分指可以用一个线性函数把多类样本分开，比如二维空间中的直线、三维空间中的平面以及高维空间中的超平面就是线性函数。线性不可分

指有部分样本用线性分类器划分时会产生分类错误。对于线性不可分样本集，可以采用神经网络来分类，神经网络就是线性分类器的升级版本。

2.1.3　代码实现

NumPy 是 Python 开源数值计算库，存储和处理大型矩阵十分方便，比 Python 自身的列表（list）结构要高效很多。本书的所有程序都基于 NumPy 实现。

为了高效地实现线性模型，可以同时计算 N 个样本的分值向量，公式如下：

$$\begin{aligned} \boldsymbol{s}_1 &= \boldsymbol{x}_1\boldsymbol{W} + \boldsymbol{b} \\ \boldsymbol{s}_2 &= \boldsymbol{x}_2\boldsymbol{W} + \boldsymbol{b} \\ &\vdots \\ \boldsymbol{s}_N &= \boldsymbol{x}_N\boldsymbol{W} + \boldsymbol{b} \end{aligned} \tag{2.5}$$

合成一个大矩阵：

$$\boldsymbol{S} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{B} \tag{2.6}$$

每个样本的分值向量组成分值矩阵 $\boldsymbol{S}$ 的一行，每个样本属性向量组成样本属性矩阵 $\boldsymbol{X}$ 的一行。矩阵 $\boldsymbol{X}$ 也称为数据矩阵，偏置矩阵 $\boldsymbol{B}$ 的每行都是偏置向量 $\boldsymbol{b}$。代码如下：

```
①   import numpy as np

     D = 784  # 数据维度
     K = 10  # 类别数
```

```
N = 128  # 样本数量

X = np.random.randn(N, D) # 数据矩阵，每行一个样本
W = 0.01 * np.random.randn(D,K)
b = np.zeros((1,K))
② scores = np.dot(X, W) + b # 广播机制
```

为了方便读者运行程序，代码会生成随机数来作为样本数据。语句①引入 numpy 模块，以后每个程序引入 NumPy 都需要采用这种写法。

细心的读者可能发现，语句②中偏置 $\boldsymbol{b}$ 是行向量，不是矩阵，严格按照矩阵运算法则，语句②是错误的，因为其维度不相同。但由于偏置矩阵 $\boldsymbol{B}$ 每行都是偏置向量 $\boldsymbol{b}$，如果非要用 $\boldsymbol{B}$，就会很麻烦，需要把 $\boldsymbol{b}$ 扩展成矩阵，但实质内容只有 $\boldsymbol{b}$。因此，聪明的 NumPy 能预测到程序的意图，自动完成矩阵扩展工作，这里运用到的内部机制是广播。NumPy 广播机制对于编写高效简洁的程序十分重要，希望读者重视。NumPy 的通用函数中要求输入的数组形状（shape）是一致的。当数组的 shape 不相等时，则会使用广播机制，扩展数组使得 shape 一致，满足广播规则。

2.2 softmax 损失函数

我们了解到通过线性模型可以得到图像属于不同类别的分值向量，调整权重矩阵能改变分值向量取值，分值向量应该与训练数据集中图像的标签一致，即真实类别的分值应当取最高分值，且越高越好。

损失函数（loss function，也叫代价函数，cost function）用来评价分值向量的好坏。分值向量与真实标签之间的差异越大，损失函数值就越大，反之则越小。

2.2.1　损失函数的定义

真实类别所对应的分值应当取最高分值，分值的绝对大小不能用来直接做出判断，只有相对大小或所占比例才能体现哪个更高。softmax 损失函数采用所占比例判断，而 SVM 损失函数采用相对大小来判断。本书主要讲 softmax，想了解 SVM 的读者可参考其他资料。设分值向量 $\boldsymbol{s}=(\boldsymbol{s}_1, \boldsymbol{s}_2, \cdots, \boldsymbol{s}_k)$，因为分值可正可负，如果直接采用所占比例，对于负的分值，不能反映其相对大小。因此，运用数学技巧，把分值映射为正值，且单调递增。指数函数刚好满足此性质：$\text{exps}=\mathrm{e}^{s}$。定义归一化向量 $\text{norms}=(\mathrm{e}^{s_1}, \mathrm{e}^{s_2}, \cdots, \mathrm{e}^{s_K})/\sum_{i=1}^{K}\mathrm{e}^{s_i}$，这样每个元素取开区间 (0, 1)的值，且所有元素之和为 1。真实类别的分值应当取最高分值，且越高越好，这意味着希望真实类别的归一化向量的元素 $n\boldsymbol{s}_y$ 的取值接近 1。如果取值小于 1，需对其进行惩罚。一个直观的损失函数为 $1-n\boldsymbol{s}_y$，当取 $n\boldsymbol{s}_y$ 为 1 时，损失为 0；当 $n\boldsymbol{s}_y$ 取 0 时，损失为 1，即最大损失，满足要求。但该损失函数的惩罚力度太小，不利于后面的参数优化。我们希望当 $n\boldsymbol{s}_y$ 取 0 时，损失值很大，趋近无穷大；当 $n\boldsymbol{s}_y$ 取 1 时，损失值为 0。不难想到，负对数函数刚好满足这个条件，故损失函数采用 $-\log(n\boldsymbol{s}_y)$，我们需要最小化损失函数。把上述公式综合在一起，得到 softmax 损失函数的定义：

$$L_i = -\log(\mathrm{e}^{s_{yi}} / \sum_{j=1}^{K} \mathrm{e}^{s_j}) \tag{2.7}$$

其中 yi 是样本 i 的标签，即样本真实类别在类别集合中的序号。

这里举一个具体的例子帮助大家加深理解。假设有 3 类向量，线性模型计算得到的分值向量为 $\boldsymbol{s}=(0.35, -0.85, 1.25)$，指数分值向量为 exps = (1.42, 0.43, 3.49)，归一化向量为 norms = (0.26, 0.08, 0.66)。如果真实类别是第三类，即 $yi=3$，则损失函数值为 $-\log(0.66)=0.42$。再举一例，假设分值向量是上面例子的 5 倍即 $\boldsymbol{s}=5\times(0.35, -0.85, 1.25)$，此时归一化向量为 norms = (1.1e–02, 2.7e–05, 0.99)，损失函数值为 $-\log(0.99)=0.01$，损失值很小。注意此时损失分布很集中，即最大值 0.99 接近 1。

上面计算了一个样本的损失值，如果计算多个样本的损失值，则最终损失值取平均值。

2.2.2 概率解释

归一化向量 norms 的每个元素均大于 0 小于 1，且和为 1，所以可以将其看作归属各个类别的概率。损失函数可以看作真实类别的负对数概率，希望其最小。这个也可以通过极大似然来解释，负对数概率最小，等价概率最大，即希望真实类别样本出现的概率最大，最容易被采样到。由于真实类别的概率不可能等于 1，所以总会存在损失。只要有损失，学习就不会停止：真实类别的分值总会增加，错误类别的分值总会降低，损失值总是能够更小，这样会导致永远地学习，最

终引起过拟合。缓解这种过拟合的方法是：当真实类别的概率值大于阈值（如 0.99），则认为损失为 0，停止学习。

2.2.3　代码实现

下面实现 softmax 损失函数的计算

```
   D = 784
   K = 10
   N = 128
   # scores 是分值矩阵，每行代表一个样本
   scores = np.random.randn(N, K)
   # 样本标签
   y = np.random.randint(K, size = N)
   # 指数化分值矩阵
   exp_scores = np.exp(scores)
   # 样本归一化系数
①  exp_scores_sum = np.sum(exp_scores, axis = 1)
   # 样本真实类别的归一化分值
②  corect_probs = exp_scores[range(N), y]/exp_scores_sum
   # 负对数损失函数
   corect_logprobs = -np.log(corect_probs)
   # 平均损失
   data_loss = np.sum(corect_logprobs)/N
```

采用随机生成的分值矩阵和标签，每个样本的标签是 0 到 $K-1$ 之间的整数。语句①是行求和，因为一行代表一个样本分值向量；语句②获得样本对应类别的归一化分值，每行一个值，位置由标签 y 决定。特别需要注意的是矩阵索引方式。

2.3 优化

基于模型（如线性模型）得到分值向量，根据损失函数评价参数的好坏，如何寻找最优的参数，使损失最小？这就是最优化。

对于简单的优化问题，可以使用解析法，即直接求解出最优参数的解析解，读者在高中数学和高等数学中基本都是采用解析法，对这种方法应当很熟悉。但对于复杂的优化问题，很难得到解析解，此时一般采用数值迭代法，即首先随机初始化解 x_0，然后利用迭代公式 $x_k = \varphi(x_{k-1})$，且保证 $f(x_k) < f(x_{k-1})$ 计算数列 x_k，如果 x_k 收敛到 x^*，则 x^* 就是（局部）最优解并且 $f(x^*)$ 是最小值。迭代法需要考虑收敛速度的问题，即需要多少次迭代才能达到指定的精度要求（近似收敛到 x^*）。需要的迭代次数越少，算法收敛速度就越快，这是我们所希望的。理论上说，有 3 种收敛速度——次线性、线性和二次收敛，这 3 种收敛速度依次加快。对于非凸优化问题，最好的情况下也只能达到次线性收敛速度，收敛速度十分慢。梯度下降法是最优化的经典迭代法，牛顿迭代法是解方程的经典迭代法，下面将分别介绍这两种方法。

2.4 梯度下降法

梯度下降法是最优化的经典理论，有严格的数学理论支撑，同时也是目前训练神经网络最常用和最有效的优化方法，请读者给予足够的重视。

2.4.1 梯度的解析意义

读者首先需要对梯度建立全面的理解，包括解析、几何和物理。下面介绍解析意义。先从最简单的一元函数开始，根据泰勒一阶展开式：

$$f(x) \approx f(x_0) + (x - x_0) f'(x_0) \tag{2.8}$$

需要注意的是，公式在 x_0 的邻域内成立，也就是 x 需离 x_0 很近。最小化 $f(x)$，x 必须满足：

$$f(x) \approx f(x_0) + (x - x_0) f'(x_0) < f(x_0) \tag{2.9}$$

即 $(x - x_0) f'(x_0) < 0$，两个数乘积为负，必为异号：$(x - x_0) = -\eta f'(x_0)$，得 $x = x_0 - \eta f'(x_0)$，其中 η 是正的比例系数，在梯度下降法中称为学习率。改变变量下标，就得到迭代公式：

$$x_k = x_{k-1} - \eta f'(x_{k-1}) \tag{2.10}$$

收敛条件是 $f'(x_k) = 0$，即 x_k 是（局部）极值点，达到最小化目的。

现在推广到多元函数，多元函数的泰勒一阶展开式为：

$$\begin{aligned} f(x^1, x^2, \cdots, x^n) \approx f(x_0^1, x_0^2, \cdots, x_0^n) &+ (x^1 - x_0^1) f'_{x_0^1}(x_0^1, x_0^2, \cdots, x_0^n) \\ &+ (x^2 - x_0^2) f'_{x_0^2}(x_0^1, x_0^2, \cdots, x_0^n) + \cdots \\ &+ (x^n - x_0^n) f'_{x_0^n}(x_0^1, x_0^2, \cdots, x_0^n) \end{aligned} \tag{2.11}$$

其中 $f'_{x_0^i}(x_0^1, x_0^2, \cdots, x_0^n)$ 是对变量 x_0^i 的偏导数。$(x_0^1, x_0^2, \cdots, x_0^n)$ 看作点 x_0，$(x^1, x^2, \cdots, x^n)$ 看作点 x，则向量 $x - x_0 = ((x^1 - x_0^1), (x^2 - x_0^1), \cdots, (x^n - x_0^n))$。令向量 $\boldsymbol{g}(x_0) = (f'_{x_0^1}(x_0), f'_{x_0^2}(x_0), \cdots, f'_{x_0^n}(x_0))$，则展开式简化为 $f(x) \approx$

$f(x_0)+\boldsymbol{g}(x_0)^{\mathrm{T}}(x-x_0)$。因为内积 $\boldsymbol{g}(x_0)^{\mathrm{T}}(x-x_0)=|\boldsymbol{g}(x_0)||x-x_0|\cos(a)$，其中 a 是这两个向量的夹角，只要夹角大于 90 度，$f(x)$就会降低，但要最小化$f(x)$，向量 $x-x_0$ 必须与向量 $\boldsymbol{g}(x_0)$方向相反，即 $x-x_0=-\eta\boldsymbol{g}(x_0)$。改变变量下标，就得到迭代公式：

$$x_k = x_{k-1} - \eta \boldsymbol{g}(x_{k-1}) \tag{2.12}$$

对比一元和多元迭代公式，发现形式完全一致。定义 $\boldsymbol{g}(x)=(f'_{x^1}(x), f'_{x^2}(x), \cdots, f'_{x^n}(x))$ 为多元函数 $f(x)$ 的梯度，由于迭代公式中梯度前有负号，且使函数值下降，故该法称为梯度下降法。其中梯度是向量，有方向和大小，但由于学习率的调整，所以梯度大小没有意义，只有其方向才对最优化有影响。

一定要注意，迭代公式中，点 x_k 位于点 x_{k-1} 的邻域内，泰勒展开式才能成立，为了使 $\eta\boldsymbol{g}(x_{k-1})$ 的绝对值充分小，学习率 η 必须充分小，才能保证 $f(x_k)$ 随着迭代次数 k 的增大而减小。学习率太小，会使收敛速度过慢，即每次 $f(x_k)$ 下降的值很少，需要很多次迭代才能获得比较好的最小值。如果学习率过大，收敛速度虽然会加快，但有发散风险，可能找不到最小值。如何选择大小合适的学习率一直是机器学习的中心问题，目前还没有好的解决方案，只能采用试错法。需要注意的是，当函数收敛到 $f'(x_k)=0$ 的点 x_k，由于泰勒展开式的局部性，这点不一定是全局最小点，有可能是局部最小点或鞍点。具体会收敛到哪种点，这和初始点 x_0 密切相关。目前初始点只能随机初始化，所以为了能得到更小的函数值，一般会进行多次优化，每次都会产生不同的随机初始点，选择最小的函数值作为函数最小值。

2.4.2 梯度的几何意义

函数 $f(x)$ 在点 x 增长最快的方向是梯度方向，与函数等值面的法向量一致，由数值较低的等值面指向数值较高的等值面。具体地，在等高线地形图上，梯度方向就是垂直于等高线最密的地方。举个实例，假设二元函数 $f(x, y) = (x-x_0)^2+(y-y_0)^2-c = 0$，等值线是圆周（$c$ 取不同值），梯度为 $g(x, y) = 2 \times ((x-x_0), (y-y_0))$，梯度方向为从圆心 (x_0, y_0) 指向圆上的点 (x, y)，是圆在点 (x, y) 处的法线方向，与圆周（等值线）垂直。

2.4.3 梯度的物理意义

举一个三维曲面的例子，如三维地形，在曲面的任一点放置一个静止的小球，在重力作用下小球会向下滚动，那么初始时刻会朝哪个方向滚动呢？答案就是负梯度方向！小球会一直沿着负梯度方向滚动，在摩擦力和重力的共同作用下，最终停在一个凹谷处！凹谷就是这个三维曲面的（局部）极小值！

传热问题同样是这样，物体热量从高温部位传导到低温部位，最后到达最低温部位，传导方向就是沿着温度的负梯度方向。

2.4.4 梯度下降法代码实现

下面给出梯度下降法的相关代码，以一元函数为例：

```
alpha = 1
epslon = 0.5
iter_num = 100
x0 = 1
```

```
def f(x):
    return alpha*x**2/2

def df(x):
    return alpha*x

def GD_update(x):
    return x - epslon*df(x)

x = x0
for k in range(iter_num):
    x = GD_update(x)
    print(k, x, f(x), df(x))
```

读者可以尝试不同的学习率 `epslon`，来感受梯度下降法的实际效果，当然也可以改变函数 `f(x)`。

2.5 牛顿法

梯度下降法是根据函数的一阶泰勒展开式推导的，能不能用二阶泰勒展开式呢？答案是肯定的，这就是牛顿法。先从最简单的一元函数二阶泰勒展开式开始：

$$f(x) \approx f(x_0) + (x - x_0) f'(x_0) + \frac{1}{2}(x - x_0)^2 f''(x_0) \tag{2.13}$$

我们知道在极值点，函数导数为 0。那么对式(2.13)求导，并令导数为 0，注意变量是 x，得 $f'(x_0) + (x - x_0) f''(x_0) = 0$，移项得 $x = x_0 - f'(x_0) / f''(x_0)$，改变下标，得到迭代公式：

$$x_{k+1} = x_k - f'(x_k) / f''(x_k) \tag{2.14}$$

牛顿迭代公式利用了二阶导数，对于二次函数，只需迭代一步就能获得极小解 x^*，具有二次收敛速度。对于非二次函数，也具有很快的收敛速度，这是牛顿法的主要优点。此外，迭代公式中不存在学习率，这也是牛顿法的优点之一。牛顿法的最大缺点是需要计算二阶导数。在一元函数时，这个缺点不明显，而 N 元函数的“二阶导数”是海森矩阵，矩阵大小是 $N\times N$，当 N 很大时，计算和存储海森矩阵就是个大问题。特别是对于卷积网络来说，如果 N 是 100 万（属于中型网络），每个元素是双精度（8 个字节），则存储海森矩阵需 7450 GB，远超当前主流存储器的容量。目前，对牛顿法最先进的改进算法是 LBFGS，这种方法对海森逆矩阵进行近似计算，并采用特征向量计算海森矩阵而不是存储整个矩阵，这样不仅极大地加快了海森逆矩阵的计算速度，还减小了存储量。目前，这个研究方向是卷积网络优化研究的一个重要课题。但应当指出，目前使用最广的还是基于梯度下降法的一些改进算法，这部分内容会在第 5 章详细介绍。

2.6　机器学习模型统一结构

机器学习包括 3 个部分：参数化的映射模型，将原始特征属性映射为类别分值向量；损失函数，根据类别分值向量和训练集标签的相符程度，衡量参数好坏；最优化，寻找最优参数，使损失函数值最小。

本书主要涉及的映射模型有线性模型、神经网络和卷积神经网络，讲解的损失函数为 softmax 损失函数，并且介绍基于梯度下降法的误

差反向传播优化方法。

基于梯度下降法的误差反向传播优化方法必须有损失函数，而损失函数的定义中必须有样本标签，因为损失函数的任务就是衡量分值向量和样本标签的一致性，所以这种学习方法也叫作有监督学习。有监督学习的最大问题是必须提供足够的带标签的训练集，而建立这样的训练集需要大量人力和物力。因此，现在业界开始研究弱监督学习或无监督学习。那么，为什么需要如此多带标签的训练集呢？这是因为卷积网络的权重参数的数量很大，达百万或千万，为了优化这么多参数，训练集必须很大，和参数数量在一个数量级上。

梯度下降法需要计算损失函数的梯度，所以损失函数关于参数必须是可导的，或只能在有限点不可导，但需要存在次偏导。softmax 函数关于分值向量是可导的，分值向量是映射模型计算得到的，所以映射模型对参数可导。线性模型是完全可导的，神经网络和卷积神经网络几乎处处可导，只存在有限点不可导，但均存在次偏导（$\max(x, y)$ 函数存在次偏导），详见第 3 章。正因为如此，深度学习三巨头之一 LeCun 老师语出惊人："深度学习已死，可微编程万岁！"

损失函数和最优化过程这两个部分目前相对成熟，但也有很大的提升空间，特别是最优化过程。目前主流的是基于梯度下降法的误差反向传播方法，该方法的主要缺点是需要大量带标签的样本，因此如何利用无标签样本进行优化也是研究热点之一。

映射模型发展很快，特别是卷积神经网络模型的发展，可以说是日新月异。

需要指出的是，机器学习模型还包括没有可学习参数的无参数模型，如著名的最近邻（KNN）、决策树和朴素贝叶斯等方法，由于这些方法没有可学习参数，所以也就没有优化过程和损失函数。但是这些方法都有超参数，如 KNN 的近邻数、决策树的深度和最大宽度等。

2.7　正则化

正则化的目的是控制模型的学习容量，减弱过拟合的风险。不论是参数化的线性模型、神经网络和卷积神经网络等，还是无参数的决策树以及朴素贝叶斯等，机器学习模型均存在过拟合现象。过拟合的外在表现形式是：模型在训练集上的准确率明显高于在测试集上的准确率，即模型对训练集学得比较好，而测试集学得比较差。所以在实践中，经常利用这个性质来判断模型是否发生过拟合，如果过拟合，就需要增加正则化强度。

举个例子，现在要拟合平面上的 3 个点，而二次函数能恰好拟合这 3 个点，所以这个任务的理论容量只需二次函数。假设现在用三次函数进行拟合，显然三次函数也能完美拟合这 3 个点。但是存在一个很严重的问题：解不唯一，即存在无穷个三次函数能完美拟合这 3 个点。对于学习任务来说，如果模型存在多种可能的解，则说明模型的学习容量过高了。我们到底该选择哪一个解，这个解相比其他解的优势在哪里呢？这个问题目前难以回答，普遍采用的原则是“奥卡姆剃刀”。它是一种常用的自然科学研究中最基本的原则，即“当有多个假设与观察一致时，选择最简单的那个”，简单就是美。如果采用这个原

则，那么对于三次函数，哪种形式更简单呢？这显然不存在一个简单答案，需借助其他机制才能解决。假设认为简单的三次函数就是系数的平方和最小，即系数的 2 范数最小，则我们就能从无穷个三次函数中选择一个最“合理”的函数。这个额外的对系数 2 范数最小化的要求，就是我们对模型的偏好，也就是 L2 正则化。所以，正则化可以看作我们对模型的偏好。如果偏好设置合理（符合样本真实规律），就能选到更好的解。除了系数的平方和最小，还可以要求系数的绝对值之和最小，这就是 1 范数，L1 正则化；要求系数不为 0 的数量最少，这就是 0 范数，L0 正则化。这 3 种范数对应 3 种正则化技术，是最常见的正则化技术。对于上面三次函数拟合问题，应该是 0 范数偏好最合理，因为这会使三次项系数会优化为 0，求得的最优函数就是二次函数。

再次强调，在实践中，一般采用容量大的模型进行学习，然后通过正则化来控制过拟合。通过上面的例子可以看出，容量大的模型存在更多的可能解。通过梯度下降法能比较容易找到这些解，然后通过正则化偏好选择其中一个。相比之下，容量小的模型的解可能很少，因此要找到质量高的解很难。

增加模型容量很简单。对于神经网络和卷积网络，只需增加深度或宽度，我们一般优先增加深度。

2.7.1 范数正则化

对于生活中的实际问题，我们很难知道理想模型的形式，往往不能判断哪种范数更好，因此，实际中最常用的是 L2 正则化技术。

L2 正则化技术，即在损失函数中增加权重 $\boldsymbol{w}$ 的平方和 $\Sigma 1/2\lambda \boldsymbol{w}^2$，其中正常数 λ 是正则化系数，λ 越大说明正则化强度越大。L2 正则化最大的优点是：权重的绝对值变小，没有哪个权重的比重过大（即各个权重的大小差不多），这样输入的每个属性通过与权重相乘，都能对分类产生影响。

L1 正则化技术，即在损失函数中增加权重 $\boldsymbol{w}$ 的绝对值和 $\Sigma\lambda|\boldsymbol{w}|$，其中正常数 λ 是正则化系数，λ 越大说明正则化强度越大。L1 正则化的最大优点是：权重变得稀疏，即权重为 0 的数量比较多。注意，这里是准确地为 0，而不像 L2 正则化趋向 0。L1 正则化可以达到特征选择的目的，即权重为 0 的特征对分类没有影响，可以去除。

L0 正则化技术的主要目的是使权重尽可能为 0,使权重变得稀疏。但由于它不可导，所以实践中几乎不采用。同时，由于 L1 正则化也有稀疏特性，所以常采用 L1 正则化来代替 L0 正则化。

为了使读者对范数正则化有一定的感性和理性认识，我们对损失函数进行简化分析。假设只有一个权重，数据损失在局部可近似为二次曲线（泰勒展开）。

L2 正则化的总损失函数为：

$$\text{loss} = 1/2\alpha(w - w_0)^2 + 1/2\lambda w^2 (\alpha, \lambda > 0) \tag{2.15}$$

对上式求导并将其等于 0，得最优权重为 $w = \dfrac{1}{1+\dfrac{\lambda}{\alpha}} w_0$，没有正则化时的最优权重为 $w' = w_0$，可见 $|w| < |w'|$，即权重收缩，趋向于 0。进一

步分析，当$\frac{\lambda}{\alpha} \ll 1$时，最优权重收缩得小，受正则化影响小；当$\frac{\lambda}{\alpha} \gg 1$时，最优权重收缩得大，受正则化影响大。$\alpha$ 的物理意义是权重 w 对损失函数的敏感程度，α 越大，表示权重对损失的影响越大，权重越重要。上面的分析表明：重要的权重受正则化影响小，不重要的权重受正则化影响大，其值会趋向 0，但不等于 0。

L1 正则化的总损失函数为：

$$\text{loss} = 1/2\alpha(w - w_0)^2 + \lambda |w| (\alpha, \lambda > 0) \tag{2.16}$$

由于式(2.16)不可导，因此求最小值比较困难，而式(2.16)显然关于 w_0 对称，故假设 $w_0>0$，可得最优权重为 $w = \max(w_0 - \frac{\lambda}{\alpha}, 0)$，没有正则化时的最优权重为 $w' = w_0$，可见 $w < w'$，即权重变小。进一步分析，当$\frac{\lambda}{\alpha} \geqslant w_0$时，最优 $w = 0$；当$\frac{\lambda}{\alpha} < w_0$时，最优 $w = w_0 - \frac{\lambda}{\alpha}$。上面分析表明：不重要的权重（$\alpha$ 小）受正则化影响大，其值等于 0；重要的权重受正则化影响小，但更接近 0（向 0 移动了$\frac{\lambda}{\alpha}$）。当 $w_0<0$ 时，分析结论是一致的，即重要的权重靠近 0，不重要的权重等于 0。

综上分析，范数正则化后，最优权重会受到影响，即重要权重向 0 靠近，不重要权重变为或趋近 0。总之，权重的绝对值都会变小。可以说，范数正则化偏好小的权重，认为小权重的模型更简单。因为正则化强度越大（λ 越大），最优权重受到的影响越大，所以数据损失部分会变大。

需要注意的是，与权重不同，偏置不需要正则化，因为偏置不与特征相乘，不能控制特征的影响强度。

L2 正则化的代码实现如下：

```
reg = 10**(-4)
reg_loss = 0.5*reg*np.sum(W*W)
```

范数正则化只适用于参数化模型，对于如何正则化无参模型（如决策树），读者可自行查阅相关资料。

2.7.2　提前终止训练

当模型采用迭代法进行优化时，如果一直迭代下去，训练集的错误率会缓慢下降，但验证集的错误率会先下降再上升，这表明模型发生了过拟合。所以，我们可以在验证集错误率最低的时候终止训练，利用此时的参数作为模型的最终参数,可以期望获得更好的测试表现。从控制过拟合的观点看，训练次数也是模型的超参数，这个超参数在验证集上具有 U 形的性能曲线。训练次数只需要对模型训练一次，就可以尝试很多值的超参数。提前终止十分有效，所以是最常用的正则化方法。提前终止法的缺点是需要验证集，而此时验证集的样本就不能用来训练模型，这导致样本的利用率不高。

提前终止法是如何进行正则化的？从表面上看，我们并没有像范数正则化那样对参数进行任何约束。其实，提前终止法把参数限制在初始参数值的小邻域内，即用学习率 ε 进行了 T 次迭代。假设梯度有界，则 εT 就能衡量参数从初始值到达的空间大小。由于参数初始值一般在 0 附近，这样就相当于限制参数为小值，偏好小的参数，和范数正则化一致。

2.7.3 概率的进一步解释

softmax 分类器的归一化向量为每个类别提供了概率，该向量的最大元素越大，分类结果正确的“可能性”就越大。为什么说“可能性”呢？这是因为分值的具体取值受正则化参数 λ 控制。如果参数 λ 变大，权重 w 就会更小、更分散，分值也会变得更小，这样归一化向量的元素值会变得更均匀。随着参数 λ 不断增大，权重就会越来越小，最后输出的概率会接近于均匀分布，类似 2.2.1 节中的例子。这就是说，softmax 分类器输出的概率的绝对大小难以直观解释，只提供一种对分类结果的相对自信。

第 3 章

神经网络

神经网络是对线性模型的升级，使之能对线性不可分的训练集达到好的分类效果，同时也是理解卷积神经网络的基础，其核心是引入非线性激活函数和多层结构。

3.1 数学模型

在线性模型中，我们利用矩阵乘法获得由图像像素到分值向量的映射，图像的像素被拉伸成一个输入行向量，参数是矩阵，输出是分值向量，其维数是类别数量。比如，在 MNIST 数据库中，输入是一个 $[1\times784]$ 的行向量，参数是一个 $[784\times10]$ 的矩阵，输出是 10 维的分值行向量。

线性模型只能对线性可分的训练集达到较好的分类效果，那么怎么对其升级，使之能对线性不可分的训练集也达到好的分类效果呢？如果对线性模型的计算过程进行抽象，设输入行向量为 $\boldsymbol{x}$，参数矩阵

为 $\boldsymbol{W}$，分值向量为 $\boldsymbol{y}$，则 $\boldsymbol{y}=\boldsymbol{x}\boldsymbol{W}$。这个公式从线性代数的角度看，就是线性变换，即把向量 $\boldsymbol{x}$ 通过变换矩阵 $\boldsymbol{W}$ 变换为向量 $\boldsymbol{y}$。从线性变换的角度对线性模型进行升级，就很容易理解。向量可以看作高维空间的点,线性变换就是把一个维度空间的点变换到另一个维度空间的点。线性模型只进行了一次线性变换，那么能不能通过多次线性变换对其升级呢？尝试进行两次线性变换，第一次变换矩阵为 $\boldsymbol{W}_1$，输出为 $\boldsymbol{h}$；第二次变换矩阵为 $\boldsymbol{W}_2$，输出为 $\boldsymbol{y}$。则公式为：

$$\begin{aligned}\boldsymbol{h}&=\boldsymbol{x}\boldsymbol{W}_1\\\boldsymbol{y}&=\boldsymbol{h}\boldsymbol{W}_2\end{aligned}\tag{3.1}$$

合并为：

$$\boldsymbol{y}=\boldsymbol{x}\boldsymbol{W}_1\boldsymbol{W}_2\tag{3.2}$$

根据矩阵运算法则，令 $\boldsymbol{W}=\boldsymbol{W}_1\boldsymbol{W}_2$，则式 (3.2) 变为 $\boldsymbol{y}=\boldsymbol{x}\boldsymbol{W}$，仍然是一个线性变换，只是最终的变换矩阵是两个变换矩阵的乘积，并没有达到升级的目的。怎样才能利用多次变换，又不会最终合并成一个变换呢？这里最关键的是引入非线性。在上面的过程中，第一次变换的输出 $\boldsymbol{h}$，它直接作为第二次变换的输入，如果对输入 $\boldsymbol{h}$ 先进行非线性变换后，再将其作为第二次变换的输入，则最终变换就不会合并成一个变换，达到升级的目的。加入非线性变换的公式为：

$$\begin{aligned}\boldsymbol{a}&=\boldsymbol{x}\boldsymbol{W}_1\\\boldsymbol{h}&=f(\boldsymbol{a})\\\boldsymbol{y}&=\boldsymbol{h}\boldsymbol{W}_2\end{aligned}\tag{3.3}$$

注意非线性函数 f 作用在向量 $\boldsymbol{a}$ 上，是逐元素进行运算的，而不

是对向量 $\boldsymbol{a}$ 的整体运算。不同的非线性函数 f 对最终的学习效果有很大影响。

理论上说，我们可以进行很多次线性变换，每次线性变换的输出向量都需要进行非线性变换。注意，最后一次的线性变换不需要进行非线性变换，因为最后的输出 $\boldsymbol{y}$ 多用于表示分值（分类），可以是任意的实数值，或者实数值的目标值（回归）。例如，三次变换的公式为：

$$\begin{aligned} \boldsymbol{h}_1 &= f(\boldsymbol{x}\boldsymbol{W}_1) \\ \boldsymbol{h}_2 &= f(\boldsymbol{h}_1\boldsymbol{W}_2) \\ \boldsymbol{y} &= \boldsymbol{h}_2\boldsymbol{W}_3 \end{aligned} \tag{3.4}$$

在一般情况下，同一个模型中的非线性函数 f 取相同的函数形式。在神经网络术语中，$\boldsymbol{x}$ 是输入层，$\boldsymbol{y}$ 称为输出层，中间变换层 $\boldsymbol{h}_1$ 和 $\boldsymbol{h}_2$ 称为隐含层，非线性函数 f 称为激活函数。存在多少次线性变换，就称为多少层神经网络，所以上面的网络是三层网络。网络的深度是网络的层数，隐含层向量的维数称为网络宽度，向量元素称为神经元。

3.2　激活函数

激活函数是神经网络的关键，用于对输入向量进行逐元素计算。sigmoid 激活函数的数学公式是 $\sigma(x) = 1/(1+\mathrm{e}^{-x})$，函数图像如图 3.1 左图所示，把输入的实数值“压缩”到 (0, 1) 开区间：负数端趋近 0，正数端趋近 1。sigmoid 函数的应用曾经非常广泛，因为它能很好地解释神经元的激活：从完全不激活（输出 0）到完全激活（输出 1）。

但现在 sigmoid 函数很少使用了，这是因为它有一个重大缺点：饱和特性使梯度消失。当输入比较大的负数或比较大的正数时，函数值接近 0 或 1，处于饱和状态，此时梯度几乎为 0，这导致网络不能进行有效学习，第 6 章会详细解释这一点。现在你只需记住，网络在激活函数梯度接近 0 的区域学习困难。

tanh 非线性函数的图像如图 3.1 右图所示，将实数值压缩到 (–1, 1) 开区间。虽然它和 sigmoid 函数一样，也存在饱和区域，但与 sigmoid 神经元不同的是，它的输出是零中心的，因此 tanh 会比 sigmoid 的学习效果更好。tanh 是放大平移后的 sigmoid，这两个函数的关系如下：

$$\tanh(x) = 2\sigma(x) - 1$$

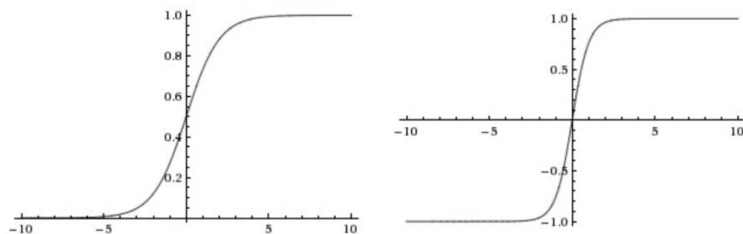


图 3.1　sigmoid 函数（左）和 tanh 函数（右）的图像

ReLU（Rectified Linear Unit，修正线性单元）函数的数学公式是 $f(x)=\max(0,x)$，其图像如图 3.2 所示。它与线性函数 $f(x)=x$ 十分相似，只是输入小于 0 的部分都输出为 0，函数图形是过原点的折线，十分接近线性函数。这几年 ReLU 非常流行，相比于 sigmoid 和 tanh 函数，ReLU 采用随机梯度下降法进行优化时，收敛速度更快，如 2012 年的 AlexNet 网络表明 ReLU 收敛速度是 tanh 的 6 倍左右。这可能是因为

该函数在输入大于 0 时的梯度为 1。sigmoid 和 tanh 函数需要指数运算，而 ReLU 只需与阈值（0）进行比较，运算更快。其缺点是，当 ReLU 的输入在小于 0 的区域时，函数值永远为 0，梯度为 0。同样是饱和区域，这会导致学习困难。但是当学习率较小时，能较好地克服该缺点。

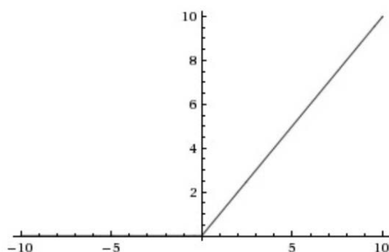


图 3.2　ReLU 函数的图像

ReLU 存在各种改进版本,都是为了克服 ReLU 在输入小于 0 时函数值恒为 0 的缺点，使其不恒为 0，具有一定梯度值。当输入大于 0 时，保持不变。

Leaky ReLU 是 ReLU 的一种改进版本，其公式为：

$$f(x) = I(x<0)(\alpha x) + I(x \geqslant 0)(x)$$

其中 α 是一个小的正常量，$I(x)$ 是指示函数，括号内的逻辑表达式为真时，输出 1，否则输出 0。函数 $f(x)$ 在输入小于 0 时会给出一个很小的梯度值 α，比如 0.01。这个激活函数有时表现不错，但并不稳定。

PReLU（参数化 ReLU）把负区间上的斜率当作可学习的参数，而不是固定的 0 或小常数，由此增大学习容量。但是有研究指出，该函数并不是在所有任务中都能提高学习效果。

ELU（指数线性单元）的左侧软饱和特性使 ELU 对输入变化或噪声更健壮，ELU 的输出均值接近于零，这会提高收敛速度，公式为：

$$f(x)=\begin{cases}x & \text{如果 } x \geqslant 0\\ e^{x}-1 & \text{如果 } x<0\end{cases}$$

再次强调，在同一个网络中一般使用同一种激活函数，而不混合使用多种激活函数。

在目前的实践中，推荐使用 ReLU，读者也可以尝试 Leaky ReLU 或 ELU。tanh 理论上不如 ReLU，不推荐使用 sigmoid。

3.3 代码实现

神经网络的代码和线性模型很相似，只是增加了激活函数。下面是三层神经网络计算分值向量的代码：

```
D = 784  # 数据维度
K = 10  # 类别数
N = 128  # 样本数量
dim1 = 128 # 隐含层宽度
dim2 = 36
W1 = 0.01 * np.random.randn(D, dim1)
b1 = np.zeros((1, dim1))
W2 = 0.01 * np.random.randn(dim1, dim2)
b2 = np.zeros((1, dim2))
W3 = 0.01 * np.random.randn(dim2, K)
b3 = np.zeros((1, K))

X = np.random.randn(N, D) # 数据矩阵，每行一个样本
hidden_layer1 = np.maximum(0, np.dot(X, W1) + b1)
# ReLU 激活
hidden_layer2 = np.maximum(0, np.dot(hidden_layer1, W2) + b2)
# ReLU 激活
scores = np.dot(hidden_layer2, W3) + b3 # 输出层不需要激活
```

权重和偏置参数采用随机初始化，输入样本也是随机创建的，方便读者运行程序。其中，D 是样本属性维度，K 是类别数，这两个变量对具体任务来说是常量。dim1 和 dim2 是隐含层向量的维度，是算法的超参数。注意，输出层不需要非线性激活。

从代码中能直观地看到参数数量，权重 W1 的数量为 D×dim1，权重 W2 的数量 dim1×dim2，权重 W3 的数量 dim2×K，偏置的数量分别为 dim1、dim2 和 K。由此可见，dim1 和 dim2 越大，参数数量越多，而模型学习容量越大，越容易导致过拟合。因此，为了缓和过拟合，需要正则化，最常用的正则化是权重参数的 L2 范数。

3.4　学习容量和正则化

神经网络的数学模型表现为多层嵌套的线性变换加非线性变换，在数学上称为函数族。机器学习的目的是拟合由特征向量到分值向量的映射，该映射可能是任意的，如此产生的问题是：该函数族的拟合能力如何？存在不能被神经网络拟合的函数吗？

目前，理论上已经证明，只需一个包含足够多神经元的隐含层的神经网络，就能够以任意精度逼近任意的复杂连续函数，这是一个通用的函数拟合器。但实际中的效果相对较差，这是由于隐含层需要大量的神经元，优化困难。实践表明，多层网络比单层网络好，也就说是深层网络比浅层网络好，它可以减小神经元数量，而且比较容易学习到最优参数（如采用梯度下降法）。

那么，面对一个具体的任务，如何确定最优网络结构，如隐含层的层数（0、1、2 层或更深）及每层神经元的数量。经过几十年的研究，目前理论上还没有严格的公式能指导这些超参数的设置，但得到一些经验法则。

当网络层数和每层神经元的数量增加时，网络的学习容量增加，即网络能拟合更复杂的函数（分类界面更复杂）。然而这是一把双刃剑：优点是可以分类更复杂的训练集，缺点是可能会造成过拟合。例如，二维平面上的二分类问题，训练 3 个不同的有两个隐含层的神经网络，只是隐含层神经元数量不同，如图 3.3 所示，最右边的网络正确分类了所有的训练数据，其代价是分类界面极不规则，圆圈类样本区域“侵占了”大量加号类样本区域，在测试时，效果反而会更差。最左边的网络虽然错分了 3 个圆圈样本，但分类界面光滑，把个别“深入”到加号类样本区域内的圆圈类样本看作噪声，在测试时能获得更好的泛化性能。

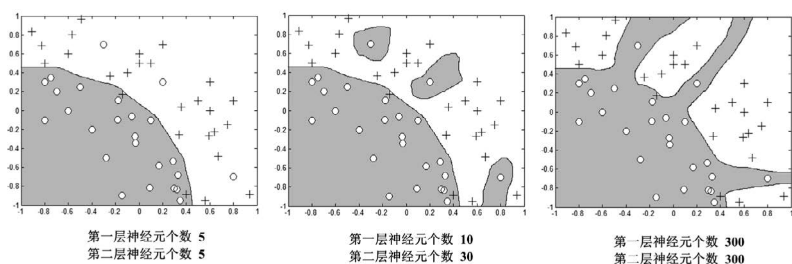


图 3.3　隐含层神经元数量和学习容量的关系。不同类别的样本用不同形状的图案表示，分类界面由训练好的神经网络做出

这个例子看起来支持采用学习容量小的网络，以防止过拟合。但实际中，如果仅是为了防止过拟合，一般不采用小网络，而是用正则

化来控制过拟合。

这主要是因为对于小网络，当优化算法采用梯度下降法等局部方法时，优化算法比较容易收敛到损失值比较大的局部极小值，导致分类准确率过低。而对于大网络，优化算法不管收敛到哪个局部极小值，这些局部极小值的损失值均比较低，而且分类准确率都较高。实践表明：小网络对参数初始值很敏感，好的初始值能收敛到低损失值，坏的初始值会收敛到很高的损失值；而大网络对参数初始值不敏感，均能收敛到较小的损失值。在实践中，一般采用多次随机初始化参数，观察损失值分布，如果方差较小，说明网络规模较大；如果方差过大，说明网络规模过小，这时需要增加网络规模。

神经网络超参数设置的技巧是尽可能地使用大网络，采用正则化来控制过拟合。多次随机初始化参数，观察损失值方差来判断网络规模是否合适，如果方差大，说明网络规模过小。

再次强调，正则化是减弱网络过拟合的好方法，如图 3.4 所示。

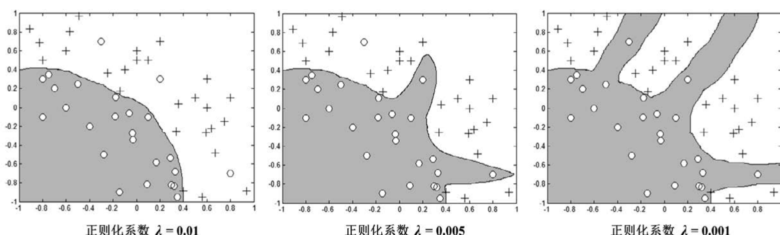


图 3.4　不同 L2 正则化强度的效果：每个神经网络都有两个隐含层，每层神经元的数量分别为 100 和 300。随着正则化强度增加，分类边界变得更加平滑，过拟合得到控制。λ 越大，表示正则化强度越大

3.5 生物神经科学基础

从前面的内容可以看出，神经网络完全可以从数学角度解释。神经网络算法最初是从生物神经系统获得启发的，目的是尽量模拟生物神经功能，但渐渐与其分道扬镳，成为一个偏工程的领域，并在机器学习领域取得良好效果。现在，我们从生物神经的角度理解上述计算。

生物神经网络最本质的特点是通过大量神经元相互连接进行信息传递。神经元是大脑的基本计算单位，人类的神经系统中大约有 860 亿个神经元，大约 10^{14} 到 10^{15} 个突触把所有神经元连接在一起，形成一个复杂的大网络。具体传递方式为：每个神经元从树突获得输入信号，然后沿着它唯一的轴突传递并产生输出信号，轴突在末端会逐渐分枝，通过突触和其他神经元的树突相连，其输出信号会成为其他神经元的输入信号。该模型可以抽象为如下的计算模型：神经元轴突传递的信号 x 基于突触强度 w（权重），与其他神经元的树突进行乘法交互。突触的强度 w 是可学习的，且控制一个神经元对另一个神经元的影响（正权重使其兴奋，负权重使其抑制）。树突将信号传递到细胞体，信号在细胞体中相加。相加之和高于某个阈值（偏置）时，神经元将会被激活，向其轴突输出一个峰值信号。神经元的激活被建模为激活函数，早期主要采用 sigmoid 函数。所以，一个神经元前向传播的计算模型的代码如下：

```
# 输入和权重是 1D 的 numpy 数组，阈值是一个数字
cell_body_sum = np.sum(inputs * weights) + bias
firing_rate = 1.0 / (1.0 + np.exp(-cell_body_sum))
# sigmoid 激活函数
```

每个神经元都对输入和权重向量进行内积，然后加上阈值，最后使用 sigmoid 激活函数。

注意，这个计算模型非常粗糙地模拟了生物神经元的核心功能。神经元计算模型与线性模型很像（都是向量内积），当神经元的激活函数值接近 1 时，认为输入样本是正类；当激活函数值接近 0 时，认为输入样本是负类，此时单个神经元就是一个线性二分类器。

神经网络是多个神经元的集合，如何组织多个神经元呢？最简单的方式是将神经元分层，每层有多个数量不等的神经元，整个网络有多层。那么，这些神经元之间如何连接呢？最简单的方式是同一层内神经元没有连接，每个神经元只与前后层神经元连接，且是全连接，即每个神经元与前后层的所有神经元都连接。这就是著名的多层前馈神经网络，其数学模型是：

$$
\begin{aligned}
\boldsymbol{h}_1 &= f(\boldsymbol{x}\boldsymbol{W}_1) \\
\boldsymbol{h}_2 &= f(\boldsymbol{h}_1\boldsymbol{W}_2) \\
\boldsymbol{y} &= \boldsymbol{h}_2\boldsymbol{W}_3
\end{aligned}
\tag{3.5}
$$

上面的模型通过函数嵌套实现神经元分层，运用矩阵乘法实现相邻层神经元的全连接，利用非线性函数实现神经元激活。模型中变量的具体含义见 3.1 节。

多层前馈神经网络是最简单的神经网络之一，除此之外还有很多，比较著名的有 RBF、ART、SOM、Elman 和 Boltzmann 等，如果读者感兴趣，可查阅相关资料。

第 4 章

卷积神经网络的结构

上一章介绍的神经网络采用分层结构，层与层之间的神经元进行全连接，本章将其称为常规神经网络。同样，卷积神经网络也是多层网络结构，层与层之间的神经元进行连接，层内神经元之间无连接。对神经元的操作也是输入和权重向量进行内积后进行非线性激活。网络最后会输出分值向量，定义损失函数，并采用梯度下降法进行优化。那么，卷积神经网络和常规神经网络有什么不同呢？

4.1　概述

发展卷积神经网络的初衷是进行图像分类。图像主要有如下 3 个特性。

- **多层次结构**：如边缘组成眼睛，眼睛和鼻子等组成脸，脸和身体等组成人。
- **特征局部性**：如眼睛就局限在一个小区域，提取眼睛特征时，只需根据这个小区域的像素提取即可。

- ❑ **平移不变性**：如不管眼睛在图像哪个位置，特征提取器都需提取眼睛特征。

根据图像的 3 个特性，卷积神经网络引入特有的先验知识——深度网络、局部连接和参数共享。

虽然卷积网络是为图像分类而发展起来的，但现在已经被用在各种任务中，如语音识别和机器翻译等。只要信号满足多层次结构、特征局部性和平移不变性 3 个特性，都可以使用卷积网络。在本章中，我们只针对图像分类任务来讲解。

4.1.1 局部连接

在常规神经网络中，每个神经元都与前一层中的所有神经元连接（全连接），但是神经元全连接方式在图像分类中既不现实也没有必要。例如，在 CIFAR-10 中，图像的尺寸是 $32\times32\times3$（宽高均为 32 像素，3 个颜色通道），如果采用全连接方式，第一个隐含层的每个神经元就有 $32\times32\times3=3072$ 个权重，这个值感觉还不是很大。但是对于更大尺寸的图像，如 $224\times224\times3$ 的图像，一个神经元包含 $224\times224\times3=$ 150 528 个权重，而隐含层肯定会包含很多神经元，那么第一个隐含层权重的数量就可能达千万量级或亿级。因此，全连接方式下，大量的参数会导致网络过拟合，而存储大量权重还需要超大内存，这一点也会限制其应用。

根据特征局部性，如果某个神经元需要提取眼睛特征，则只需针对眼睛所在的局部区域内的像素进行特征提取，不需要提取眼睛区域

外的信息，所以该神经元只需与眼睛区域进行局部连接，其他区域是不需要连接的。

4.1.2 参数共享

在图像分类中，同一物体可能会在图像的不同位置出现，例如人脸会出现在图像的任意位置，神经元必须对人脸的位置不敏感。而识别不同位置人脸的不同神经元，采用的权重应该是相同的。因为神经元学习是先通过权重和像素进行内积，再进行非线性激活实现的（同一人脸的像素相同）。这些神经元共享相同的参数，这就是参数共享。

注意，人脸仅是位置不同，即只进行了平移运动，所以称为图像的平移不变性。如果对人脸进行了旋转，特别是旋转角度比较大时，此时检测人脸所用的参数和检测未旋转的人脸一般是不同的。这和我们人类相似，读者可以自己做个实验，观察被旋转不同角度的同一张人脸照片，你会发现当角度较大时，可能就认不出此人。所以，人类视觉系统具有良好的平移不变性，但旋转不变性要差很多。卷积网络与人类视觉类似，很好地解决了平移不变性，但旋转不变性解决得不好，对旋转角度比较敏感。

4.1.3 3D 特征图

图像是二维结构，为了识别人脸，需要大量的神经元协同工作，这些神经元都与该人脸连接。要提取人脸不同的特征，必须有大量的神经元，它们从观察输入的同一局部区域提取不同特征。这样神经元

的组织方式必然是三维：高度、宽度和深度。高度和宽度决定神经元的空间尺寸，深度决定了对输入区域提取特征的维度（每个神经元提取一个特征）。例如，将 CIFAR-10 中的图像作为输入，该输入的维度是 $32 \times 32 \times 3$，32×32 是空间尺寸，3 是深度，表示同一位置有 3 个特征（即红、绿和蓝这 3 种颜色特征）；再如，卷积网络中某一层是 $7 \times 7 \times 128$，7×7 是空间尺寸，128 表示同一位置有 128 个特征。我们把神经元的 3D 排列称为 3D 特征图。3D 特征图表示为 $[H \times W \times D]$，其中 H 是高度，W 是宽度，D 是深度，宽度和高度称为空间维度。3D 特征图可以看作 D 个 2D 数据。每个 2D 数据的尺寸均是 $[H \times W]$，称为特征图，3D 特征图总共有 D 个特征图。

常规神经网络的向量可以看作 3D 特征图的特例（$1 \times 1 \times D$），即空间尺寸为 1 的 3D 特征图。卷积神经网络和常规神经网络一样，它们由层组成，每层使用可微函数将输入的 3D 特征图（向量）变换为输出 3D 特征图。

在卷积神经网络中，主要包含 3 种基本模块：卷积层、池化层和全连接层，下面将分别介绍它们。

4.2　卷积层

卷积网络采用卷积层来实现上述的局部连接和参数共享，所以卷积层是卷积网络的核心，而卷积层的核心是卷积运算。请读者思考为什么卷积运算能实现局部连接和参数共享。

4.2.1 卷积运算及代码实现

卷积网络的卷积运算和信号处理中的卷积运算不太一样，把它理解为向量内积更合适（神经元的工作机制）。在图像处理中，边缘检测算法就是利用卷积运算实现的。卷积运算是线性滤波，对于图像中的每个像素，计算以该像素为中心的局部窗口内的像素和卷积核的内积，并将其作为该像素的新值。遍历图像中的每个像素，进行上述内积操作，就完成了一次滤波，得到一个和原图像尺寸一样的“新图像”。局部窗口和卷积核的大小一样，卷积核是一个小矩阵（3×3 或 5×5），卷积运算公式为：

$$
\begin{gathered}
\boldsymbol{g}(i,j)=\sum_{m=-1,n=-1}^{m=1,n=1}\boldsymbol{f}(i+m,j+n)\boldsymbol{h}(m,n) \\
\boldsymbol{g}=\boldsymbol{f}*\boldsymbol{h}
\end{gathered}
\tag{4.1}
$$

其中 (i,j) 是中心像素的坐标，$i=1,2,\cdots,h$，$j=1,2,\cdots,w$，这里 h 是图像高度，w 是图像宽度。卷积需遍历整个图像。$\boldsymbol{f}$ 是原图像（注意它是二维矩阵），$\boldsymbol{g}$ 是“新图像”，$\boldsymbol{h}$ 是卷积核（这里卷积核的大小是 3×3），*是卷积运算符。可以看出，卷积就是滤波，所以卷积核也被称为滤波器。

当对图像边界进行卷积时，卷积核的一部分位于图像外面，无像素与之相乘，此时有两种策略：一种是舍弃图像边缘，这样会使“新图像”尺寸减小（对于 3×3 卷积核，每边会减小 1）；另一种是采用像素填充技巧，人为指定位于图像外面的像素值，使卷积核能与之相乘。像素填充主要有两种方式：0 填充和复制边缘像素。在卷积神经网络中，普遍采用 0 填充方式。

图 4.1 演示了卷积运算过程，其中输入特征图的尺寸为 4×4，采用 0 填充后尺寸为 6×6，卷积核大小为 3×3，步长为 1，则输出特征图的尺寸为 4×4，与输入特征图尺寸一致。其中，输出 0.2 是第一个局部窗口 3×3 和卷积核的内积：

$$0.2 = 0\times(-0.2)+0\times0.1+0\times(-0.1)+0\times(-0.1)+2\times(-0.1)+0\times0.2+ \\ 0\times0.3+1\times0.2+2\times0.1$$

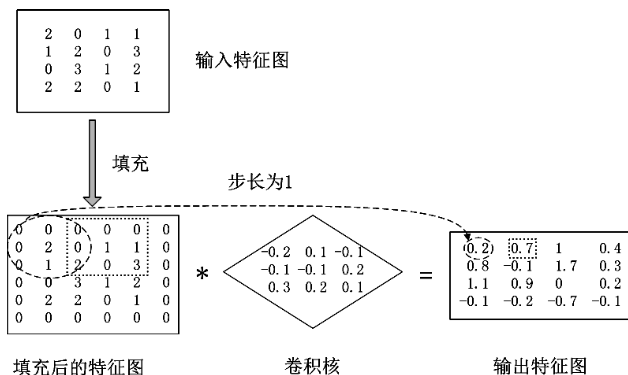


图 4.1　卷积运算示意图

在卷积运算中，卷积核的取值是核心，取值不同时，“新图像”的效果差别很大。通过卷积运算可以获得原图像的边缘、模糊图像和锐化图像等。这些著名的卷积核如下所示。

- 图像模糊的卷积核：[1 1 1; 1 1 1; 1 1 1] / 9
- 图像锐化的卷积核：[–1 –1 –1; –1 9 –1; –1 –1 –1]
- 边缘检测的卷积核：
  - Sobel：[–1 –2 –1; 0 0 0; 1 2 1]及其转置
  - Prewitt：[–1 –1 –1; 0 0 0; 1 1 1]及其转置
  - Laplace：[1 1 1; 1 –8 1; 1 1 1]

图 4.2 展示了图像边缘检测的效果，为什么 Sobel 等卷积核能检测出图像的边缘呢？这是因为在图像的边缘区域，像素值的变化剧烈，而在平滑区域，像素值基本一致。计算局部窗口的像素差能区分边缘和平滑区域，像素差大的为边缘，像素差接近 0 的为平滑区域。边缘检测的卷积核计算了窗口内像素差，其他两种卷积核请读者自行分析。



图 4.2　Sobel 卷积核检测图像边缘，左图是灰度源图像，右图是边缘强度图

下面通过代码来更清楚地了解卷积运算的细节：

```
h = 32 # 输入数据的高度
w = 48 # 输入数据的宽度
input_2Ddata = np.random.randn(h, w)
output_2Ddata = np.zeros(shape = (h, w)) # 卷积输出尺寸与输入一样

kern = np.random.randn(3, 3) # 3×3 卷积核
# kern = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]],
dtype = np.float64) # sobel 卷积核

padding = np.zeros(shape = (h+2, w+2)) # 0 填充
padding[1:-1, 1:-1] = input_2Ddata
for i in range(h):
    for j in range(w):
```

```
①            window = padding[i:i+3, j:j+3]
             # 中心像素(i,j)的局部窗口
             output_2Ddata[i, j] = np.sum(kern*window)
             # 卷积运算即内积
```

这里使用了随机生成二维输入数据 `input_2Ddata` 和 3×3 的卷积核 `kern`。注意语句 ① 中局部窗口的位置计算是以 0 填充后的图像为基准的，而公式(4.1)是以原图像为基准的，故索引有所不同。读者可以读取灰度图像，将卷积核设置为上述的卷积核，显示卷积后的新图像。新图像可以使我们获得更直观的感受，显示图像时需要注意，由于新图像的像素有正有负，需要先取绝对值再显示。

4.2.2　卷积层及代码初级实现

上面介绍的卷积运算的输入和输出数据都是 2D 特征图，而卷积网络都是 3D 特征图，如何对卷积运算进行升级，使之能处理 3D 特征图呢？

重复一遍，3D 特征图表示为 $[H\times W\times D]$，其中 H 是高度，W 是宽度，D 是深度。理解 3D 特征图是打开卷积层的钥匙，3D 特征图可以看作 D 个 2D 数据，每个 2D 数据的尺寸均是 $[H\times W]$，称为特征图，3D 特征图总共有 D 个特征图。升级做法如下：每个特征图都分别与一个卷积核进行卷积运算，这样就得到 D 个特征图，这 D 个特征图先进行矩阵相加，得到一个特征图，再给该特征图的每个元素再加一个相同的偏置，最终得到一个新的特征图。因为最终需要得到 3D 特征图，所以上述过程需进行多次，这就是一个完整的卷积层操作。从上面的过程可以看出，为了获得每个输出特征图，需要 D 个卷积核，我们把

这 D 个卷积核称为一个卷积核组，它是一个 3D 矩阵。为了获得 D 个特征图，则需 D 个卷积核组。图 4.3 演示了卷积层操作示意图。其中，输入特征图是 $3\times3\times3$，无 0 填充。卷积核组的尺寸为 $2\times2\times3$，则输出特征图的尺寸为 $2\times2\times2$。每个卷积核组有 3 个卷积核（与输入特征图数量一致），有 2 组卷积核，故输出 2 个特征图。每个输入特征图与对应的卷积核进行卷积运算，所得值相加，得到输出特征图的元素值。比如，第一个输出特征图的第一个元素 0.2，它是第一组卷积核与输入特征图的第一个局部窗口 $2\times2\times2$ 进行内积的结果。

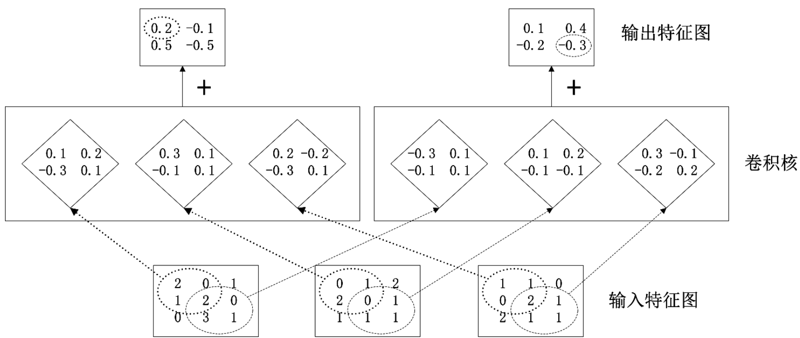


图 4.3　卷积层操作示意图

程序能代替千言万语，直观地展示卷积层运算过程的细节。下面的代码是完全按照上述描述编写的。

```
def conv2D(input_2Ddata, kern):
    (h, w) = input_2Ddata.shape # 输入数据的高度和宽度
    (kern_h, kern_w) = kern.shape # 卷积核的高度和宽度
    padding_h = (kern_h-1)//2
    padding_w = (kern_w-1)//2
    padding = np.zeros(shape = (h+2*padding_h, w+2*padding_w))
    # 0 填充
```

```
    padding[padding_h:-padding_h, padding_w:-padding_w] =
        input_2Ddata
    output_2Ddata = np.zeros(shape = (h, w)) # 输出数据的尺寸和
        输入数据一样

    for i in range(h):
        for j in range(w):
            window = padding[i:i+kern_h, j:j+kern_w]
            # 局部窗口
            output_2Ddata[i,j] = np.sum(kern*window) # 内积
    return output_2Ddata
##########################################################

h = 32 # 输入数据的高度
w = 48 # 输入数据的宽度
in_d = 12 # 输入数据的深度
out_d = 24 # 输出数据的深度
input_3Ddata = np.random.randn(h, w, in_d)
output_3Ddata = np.zeros(shape = (h, w, out_d))

(kern_h, kern_w) = (3, 3) # 或者(5, 5)
kerns = np.random.randn(out_d, kern_h, kern_w, in_d)
# 4D 卷积核
bias = np.random.randn(out_d) # 1D 偏置

for m in range(out_d): # 每一个输出 2D 数据
    for k in range(in_d): # 每一个输入 2D 数据
        input_2Ddata = input_3Ddata[:,:, k] # 第 k 个输入 2D 数据
        kern = kerns[m, :,:, k]
        output_3Ddata[:,:, m] += conv2D(input_2Ddata, kern)
        # 加上每个卷积结果
    output_3Ddata[:,:, m] += bias[m] # 每个输出 2D 数据只有
        一个偏置
```

首先定义 conv2D 函数实现常规卷积运算，注意卷积核 kern 的尺寸为奇数，一般是正方形。注意 padding 的实现细节，即 0 填充的数量。

然后定义输入和输出的 3D 特征图。注意卷积运算没有改变特征图的空间尺寸，但深度维度可能会增加。本例中，输入深度 `in_d = 12` 维，输出深度 `out_d = 24` 维。每个输出特征图需要累加输入 3D 特征图的每个 2D 特征图的卷积结果，最后加一个偏置。注意卷积核是四维矩阵，共有 `out_d` 个卷积核组，每个卷积核组的尺寸是 `[kern_h × kern_w × in_d]`，每次和输入 2D 特征图进行卷积运算的二维卷积核的取值都不相同。四维卷积核和一维偏置，就是卷积层需要学习的参数。

上面的程序有利于理解卷积层的运算，但实际的运行效率非常低，后面会实现一个高效版本的程序。

总结一下，卷积层运算需要的参数量如下。

- 卷积核四维矩阵：`out_d × kern_h × kern_w × in_d`。
- 偏置向量：`out_d`。

其中参数数量与输入和输出 3D 特征图的深度成正比，与卷积核的面积成正比。需要特别注意的是，它与特征图的空间尺寸无关。

进行上面的卷积层运算后，输出特征图的空间尺寸和输入特征图一致。这是卷积网络中最常用的卷积类型。但有时需要缩小输入特征图的空间尺寸，一般缩小为原来的四分之一，这样就不需要对输入特征图的每个元素都进行卷积运算，而是进行下采样，下采样的间隔称为步长 S。上面的代码中，步长 $S=1$，故程序不需指明步长。包含步长 S 的程序如下：

```
   def conv2D(input_2Ddata, kern, in_size, out_size,
   kern_size = 3, stride = 1):
       (h1, w1) = in_size # 输入数据尺寸
       (h2, w2) = out_size # 输出数据尺寸

       output_2Ddata = np.zeros(shape = out_size)

①      for i2,i1 in zip(range(h2), range(0, h1, stride)):
           # 输入数据进行步长
           for j2,j1 in zip(range(w2), range(0, w1, stride)):
②              window = input_2Ddata[i1:i1+kern_size,
                   j1:j1+kern_size] # 局部窗口
           output_2Ddata[i2, j2] = np.sum(kern*window) # 内积
       return output_2Ddata
   ##########################################################

   h1 = 32 # 输入数据高度
   w1 = 48 # 输入数据宽度
   d1 = 12 # 输入数据深度
   input_3Ddata = np.random.randn(h1, w1, d1)

   # 超参数
   S = 2 # 步长
   F = 3 # 卷积核尺寸
   d2 = 24 # 输出数据深度
   ####

③  P = (F-1)//2 # 填充尺寸
④  h2 = (h1-F+2*P)//S + 1 # 输出数据高度
⑤  w2 = (w1-F+2*P)//S + 1 # 输出数据宽度

⑥  padding = np.zeros(shape = (h1+2*P, w1+2*P, d1)) # 0 填充
   padding[P:-P, P:-P, :] = input_3Ddata

   output_3Ddata = np.zeros(shape = (h2, w2, d2))

   kerns = np.random.randn(d2, F, F, d1) # 4D 卷积核
   bias = np.random.randn(d2) # 1D 偏置

   for m in range(d2): # 每个输出 2D 数据
```

```
    for k in range(d1): # 每个输入 2D 数据
        input_2Ddata = padding[:,:, k] # 第 k 个输入 2D 数据
        kern = kerns[m, :,:, k] # 卷积核
        output_3Ddata[:,:, m] += conv2D(input_2Ddata, kern,
        in_size = (h1, w1), out_size = (h2, w2), kern_size = F,
            stride = S) # 加上每个卷积结果

    output_3Ddata[:,:, m] += bias[m] # 每个输出 2D 数据只有
        一个偏置
```

由于步长 `stride` 可能为 2，故 `conv2D` 函数需要知道输入和输出特征图的空间尺寸。需要注意的是，该函数的输入特征图是 0 填充后的特征图。语句①表明输出数据的步长永远为 1，输入数据的步长为参数 `stride`。语句②表明输入数据的局部窗口是连续的。语句③是计算每边 0 填充的数目，`P = (F-1)//2`，这样可以保证卷积核位于图像边缘时，刚好有足够的 0 与其进行内积。常见的 3 种卷积核尺寸对应的 `P`：`F = 3`，`P = 1`；`F = 5`，`P = 2`；`F = 1`，`P = 0`。语句④和语句⑤是计算输出特征图空间尺寸的公式：`h2 = (h1-F+2*P)//S + 1`，`w2 = (w1-F+2*P)//S + 1`，这样当 `S=1` 和 `P = (F-1)//2` 时，输出 `h2 = h1`，`w2 = w1`，卷积层没有改变特征图的空间尺寸。语句⑥进行 0 填充。注意步长 S 不影响卷积层参数数量。

算法的超参数是卷积核尺寸 F，步长 S 和输出特征图的深度 $d2$。F 不需太大，为 3 最为常见，因为卷积核参数数量与 F 的平方成正比，F 太大会导致参数数量急剧增加，运算量也急剧增加。S 最常用 1，偶尔用 2，虽然增大 S 能减小运算量，但输出特征图的空间尺寸会急剧减小，这会丢失很多信息，导致学习效果降低，因此在实践中，S 不会超过 2。深度 $d2$ 和常规神经网络的隐含层的宽度超参数类似，增大 $d2$，学习容量增大，但运算量也增加。确定最优 $d2$ 没有理论方法，实

践中采用试错法。输出特征图的深度一般大于等于输入特征图的深度，因为输出特征图的空间尺寸可能会变小，这样单个神经元观察到的局部区域会变大，所以需要提取更多的特征。

最后，卷积层操作用数学公式表示为：

$$\boldsymbol{g}(i,j)=\text{bias}+\sum_{k=1}^{k=d1}\sum_{m=-F/2,n=-F/2}^{m=F/2,n=F/2}\boldsymbol{f}_k(i'+m,j'+n)\boldsymbol{h}_k(m,n) \tag{4.2}$$

其中 $\boldsymbol{g}(i,j)$ 是输出特征图的一个元素，需以步长 S 遍历整个输入特征图：

$$i'=1,1+S,1+2S,\cdots$$
$$j'=1,1+S,1+2S,\cdots$$

4.2.3　卷积层参数总结

- 输入 3D 特征图的尺寸为 $H1\times W1\times D1$
- 3 个超参数：
  - 卷积核组的数量 K
  - 卷积核的空间尺寸 F
  - 步长 S
- 零填充数量 $P=(F-1)//2$
- 输出 3D 特征图的尺寸为 $H2\times W2\times D2$，其中：
  - $H2=(H1-F+2P)//S+1$
  - $W2=(W1-F+2P)//S+1$
  - $D2=K$
- 卷积层一共有 $K\times F\times F\times D1$ 个权重和 K 个偏置，这些超参数常见的设置是 $F=3, S=1$ 或者 $F=1, S=1$ 。

4.2.4 用连接的观点看卷积层

4.1 节指出局部连接和参数共享是卷积网络的核心概念。这显著区别于常规神经网络,因为常规神经网络采用全连接并且参数各不相同。现在看看卷积运算如何实现局部连接和参数共享。

输入和输出 3D 特征图中的每个元素称为神经元。根据上节内容,仔细分析输出神经元的激活值是怎么计算出来的?你会发现:输出神经元只观察输入神经元中的一小部分,即空间尺度上只观察卷积核内的神经元,这就是局部连接,卷积核的空间大小也叫感受野。同一特征图的所有神经元使用相同的卷积核扫描输入 3D 特征图,即参数共享。

1. 局部连接

空间维度(高度和宽度)与深度维度的连接方式是不同的:前者是局部的,后者是全连接。深度上为什么是全连接?这是因为深度方向的神经元可以看作在空间位置处提取的特征,往往需要利用所有特征进行信息加工。这种全连接方式是目前的主流做法。

CIFAR-10 图像的输入特征图的尺寸为 $32 \times 32 \times 3$,如果卷积核尺寸是 5×5,那么卷积层中每个神经元连接输入特征图中 $5 \times 5 \times 3$ 的局部区域,共 $5 \times 5 \times 3 = 75$ 个权重。

输入特征图的尺寸是 $14 \times 14 \times 128$,如果卷积核尺寸是 1×1,那么卷积层中每个神经元和输入特征图有 $1 \times 1 \times 128 = 128$ 个连接。

输入特征图的尺寸是 $7 \times 7 \times 256$,卷积核尺寸是 7×7,那么卷积

层中每个神经元和输入特征图有 $7\times7\times256=12\ 544$ 个连接。注意，此时卷积核尺寸和输入特征图空间尺寸一致，所以此局部连接就是全连接。

需要说明的是，1×1 卷积比较迷惑人，特别是对于有信号处理专业背景的人。对于二维信号，1×1 卷积没有任何意义，卷积网络特征图是三维，虽然空间维度上 1×1 卷积没有意义，但深度方向卷积有意义。比如，输入是彩色图像 $32\times32\times3$，那么 1×1 卷积就是进行三维点积。这个卷积在图像处理中有重要应用，比如彩色图像转换为灰度图像就是 1×1 卷积，公式为 $\mathrm{Gray}=\mathrm{R}\times0.299+\mathrm{G}\times0.587+\mathrm{B}\times0.114$，再加上神经元的阈值和非线性激活，就可以实现彩色图像二值化。重要的是，权重和阈值是根据特定任务的训练集学习出来的，可能会优于公式中的权重取值。公式的权重是综合各种情况给出的最普遍的数值，没有针对特定任务进行优化。

卷积网络这种局部连接方式，与常规神经网络的全连接方式完全不同，但是可以看作全连接方式的限制版本。局部连接可以看作全连接，只是把卷积核窗口外的权重参数都设置为 0。权重参数原本应该根据优化算法来进行调整，不能人为设置为 0，但是根据图像的先验知识（特征局部性），我们将它设置为 0。这就是本书第 1 章讲的模型里面要融入先验知识，这样可以提升模型的性能。

卷积网络这种局部连接方式，从特征提取的角度来看，就是输出神经元只根据局部窗口内的数据进行特征提取，同时利用深度维度上所有的数据，与窗口外的数据无关。需强调的是，输出 3D 特征图中

同一空间位置处的深度维度上的所有神经元，所观察的局部数据是一样的，所以采用的卷积核必须不同，否则提取的将是相同的特征。这也可以看作局部窗口的数据通过卷积运算和非线性激活提取了多个特征。所以 3D 特征图 $[H \times W \times D]$ 可以看作每个空间位置的元素有 D 个特征。比如彩色图像每个空间位置像素有 R、G 和 B 这 3 个特征。这种观点是后面高效程序实现的基础。

2. 参数共享

同一特征图的所有神经元使用相同的卷积核扫描输入 3D 特征图，即参数共享，能极大减小参数的数量。前面提到过，CIFAR-10 图像的输入特征图尺寸为 $32 \times 32 \times 3$，如果卷积核尺寸是 5×5，那么卷积层中每个神经元连接输入特征图中 $5 \times 5 \times 3$ 的局部区域,共 $5 \times 5 \times 3 = 75$ 个权重。如果采用步长 $S = 1$，则输出有 $32 \times 32 = 1024$ 个神经元，如果每个神经元参数都不同，则总共需要 $75 \times 1024 = 76\,800$ 个权重，仅一个特征图就需要这么多参数，而输出会有多个特征图，因此参数的数目是非常大的。

参数共享使特征图上的每个神经元都使用相同的权重。在上面的例子中，一个特征图只需 75 个权重，极大地减小了参数的数量。由于采用参数共享原则，卷积网络获得了良好的平移不变性，即图像中如果有人脸，不管人脸在图像的什么位置，卷积网络都能一致地识别出来。

由于采用了参数共享，每个特征图都可以看作只提取了一种特征。为了提高网络的学习容量，需提取多种特征，输出 3D 特征图的深度维度一般比较大。

参数共享是为了解决图像的平移不变性提出的，如果图像不具有平移不变性，则参数共享就没有意义，不同位置的神经元应该采用不同的权重来学习不同的特征。比如对准后的人脸图像，人脸一般都处于图像中心，眼睛、嘴巴等特征的位置比较固定，此时为了识别身份，就可以不使用参数共享，而只用局部连接。

参数共享的一个缺点就是相邻神经元信息高度冗余，因为它们所观察的输入局部窗口是相邻的。图像有个特点，那就是相邻像素的值比较接近，特别是在平滑区域，所以这些局部窗口的像素很相似而神经元采用相同的权重，内积计算出的激活值几乎相等，这样相邻神经元信息几乎相同，只能提供十分有限的额外信息。神经元信息冗余对于网络前几层特别明显，后面层由于经过多次非线性激活，冗余不太明显。

除了平移和旋转不变性外，图像还有尺度不变性，即人脸放大或缩小后，虽然分辨率不同了，但是理想的人脸识别算法仍应该识别出人脸。卷积运算是计算局部窗口和卷积核的内积，卷积核尺寸是固定的，不会随人脸大小的变化而变化。这样，人脸大小不同时，计算的内积是不一样的，所以神经元激活值不同，导致卷积网络对人脸尺寸敏感。因此，卷积网络在物体尺寸变化比较小时，效果很好，变化较大时，效果不好。

4.2.5　使用矩阵乘法实现卷积层运算

卷积层的基本运算是卷积核组和输入特征图的局部区域做内积，即把卷积核组和输入特征图的局部区域均拉伸为向量，然后对这两个

向量做内积运算。矩阵乘法也是两个向量做内积，因此，如果把输入特征图和所有卷积核组分别转化为矩阵，则卷积层的运算就变成两个巨大矩阵的乘法。

(1) 将输入 3D 特征图转化为矩阵 $\boldsymbol{X}$：每个局部区域均拉伸为行向量。比如，输入特征图是 $227\times227\times3$，要与尺寸为 $11\times11\times3$ 的卷积核以步长为 4 进行卷积，则首先取局部区域 $11\times11\times3$ 数据块，将其拉伸为 $11\times11\times3=363$ 的行向量。接着，扫描每一个局部区域，均拉伸为行向量。因为步长为 4，不使用 0 填充，所以输出的宽高为 (227−11)//4+1 = 55，共有 $55\times55=3025$ 个行向量，因此输出矩阵的尺寸是 3025×363。注意，由于局部区域有重叠，输入特征图中的元素在输出矩阵不同的行中有重复。

(2) 卷积核组转化为矩阵 $\boldsymbol{W}$：卷积核组拉伸成列向量。例如，尺寸为 $11\times11\times3$ 的卷积核组拉伸为 $11\times11\times3=363$ 的列向量，如果输出特征图的深度是 96，则有 96 个列向量，就生成一个矩阵 $\boldsymbol{W}$，尺寸为 363×96。

(3) 现在卷积层操作和矩阵乘法 np.dot($\boldsymbol{X}$, $\boldsymbol{W}$) 是等价的，即得到每个卷积核组和每个局部区域数据的内积。在本例中，输出矩阵是[3025×96]。

(4) 矩阵重新变为输出 3D 特征图的尺寸 $55\times55\times96$，每行就是输出 3D 特征图的一个深度向量，或每列就是输出 3D 特征图的一个特征图。

这个方法的缺点是占用内存较多，因为输入特征图中的元素在

X 中会被复制很多次，优点是矩阵乘法实现起来非常高效（常用的 BLAS API）。

图 4.4 展示了使用矩阵乘法实现卷积层运算的示意图，输入 3D 特征图是 $3\times3\times3$，卷积核组是 $2\times2\times3$，步长 $S=1$，0 填充 $P=0$，卷积核组的数量 $K=2$，故输出是 $2\times2\times2$。输入 3D 特征图的每个局部区域 $2\times2\times3$ 数据均拉伸为行向量 $[1\times12]$，总共有 4 个局部区域，得矩阵 $\boldsymbol{X}=[4\times12]$。每个卷积核组拉伸为列向量 $[12\times1]$，有两个卷积核组，得矩阵 $\boldsymbol{W}=[12\times2]$，然后进行矩阵乘法 $\boldsymbol{XW}$，得到输出矩阵 $[4\times2]$，每行就是一个深度向量，转变为输出 3D 特征图 $2\times2\times2$。

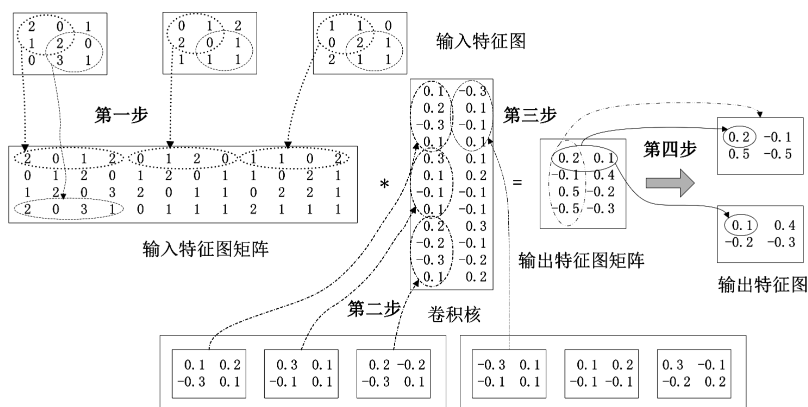


图 4.4　使用矩阵乘法实现卷积层运算的示意图

4.2.6　批量数据的卷积层矩阵乘法的代码实现

为了高效利用矩阵乘法，一般会批量进行图像的卷积运算。令批量为 B，输入数据就是 4D 数据 $[B\times H\times W\times D]$。每个 3D 特征图都转变为二维大矩阵，然后按行堆叠在一起，形成一个超级大矩阵。本例中，

如果有 $B=10$ 个输入特征图，则超级大矩阵 $\boldsymbol{X}$ 的尺寸是 $30\,250\times 363$。第 (2) 步和第 (3) 步不变，第 (4) 步要变为 4D 输出数据 $10\times 55\times 55\times 96$。

算法的核心是实现 4.2.5 节所述方法的第 (1) 步，把局部窗口数据拉伸为行向量，然后遍历每个特征图的每一个局部窗口，使这些行向量按行堆叠在一起即可。代码如下：

```
    filter_size = 3
    filter_size2 = filter_size*filter_size
    stride = 1
①   padding = (filter_size -1)//2

    (batch, in_height, in_width, in_depth) = (8, 32, 48, 16)
②   in_data = np.random.randn(batch, in_height, in_width,
    in_depth)

③   out_height = (in_height - filter_size + 2*padding)//stride + 1
④   out_width = (in_width - filter_size + 2*padding)//stride + 1
    out_size = out_height*out_width

⑤   matric_data = np.zeros( (out_size*batch,
    filter_size2*in_depth) )

⑥   padding_data = np.zeros((batch, in_height + 2*padding,
    in_width + 2*padding, in_depth) )
    padding_data[:, padding : -padding, padding : -padding, :] =
    in_data

⑦   height_ef = padding_data.shape[1] - filter_size + 1
⑧   width_ef = padding_data.shape[2] - filter_size + 1

⑨   for i_batch in range(batch):
⑩       i_batch_size = i_batch*out_size
⑪       for i_h, i_height in zip(range(out_height),
            range(0, height_ef, stride)):
⑫           i_height_size = i_batch_size + i_h*out_width
```

```
⑬            for i_w, i_width in zip(range(out_width), range(0,
                 width_ef, stride)):matric_data[i_height_size +
                     i_w, :] = padding_data[i_batch,
                     i_height : i_height + filter_size,
⑭                    i_width  : i_width + filter_size, :].ravel()
```

先设置算法超参数：卷积核尺寸 filter_size 和步长 stride。语句 ① 计算 0 填充 padding；语句 ② 随机生成 4D 的输入特征图；语句 ③ 和语句 ④ 计算输出特征图的高度和宽度；语句 ⑤ 分配输出大矩阵的存储空间，注意行数量为 out_size*batch；语句 ⑥ 进行 0 填充；语句 ⑦ 和语句 ⑧ 计算卷积运算以步长 stride 滑动时，在输入数据体上最大能滑动到的位置；语句 ⑨ 遍历每个输入 3D 特征图；语句 ⑩ 计算第 i_batch 个 3D 特征图的首个局部窗口数据的行位置；语句 ⑪ 遍历每一行；语句 ⑫ 计算第 i_h 行首个局部窗口数据的行位置；语句 ⑬ 遍历每一列；语句 ⑭ 获取局部窗口数据，并使用 ravel 方法将其拉伸为 1D 向量，赋值给对应的行。

第 (2) 步，卷积核组拉伸为列向量。实际上并不需要事先生成四维的卷积核，直接生成二维卷积核矩阵即可，代码如下：

```
out_depth = 32
weights = 0.01 * np.random.randn(filter_size2*in_depth,
    out_depth)
bias = np.zeros((1, out_depth))
```

这里设置超参数 out_depth，生成小的随机数初始化权重矩阵。注意矩阵有 out_depth 列。偏置初始化为 0。

第 (3) 步，矩阵相乘和 ReLU 非线性激活，代码如下：

```
filter_data = np.dot(matric_data, weights) + bias # 广播机制
filter_data = np.maximum(0, filter_data) # ReLU 激活
```

第 (4) 步，把 filter_data 的每一行数据转变为输出 4D 特征图对应位置的深度维度的数据。代码如下：

```
①   out_data = np.zeros((batch, out_height, out_width, out_depth))

②   for i_batch in range(batch):
③       i_batch_size = i_batch*out_size
④       for i_height in range(out_height):
⑤           i_height_size = i_batch_size + i_height*out_width
⑥           for i_width in range(out_width):
⑦               out_data[i_batch, i_height, i_width, :] =
                    filter_data[i_height_size + i_width, :]
```

语句 ① 分配输出 4D 特征图的存储空间；语句 ② 遍历每个输出 3D 特征图；语句 ③ 计算第 i_batch 个 3D 特征图的首行位置；语句 ④ 遍历每一行；语句 ⑤ 计算第 i_height 行的首行位置；语句 ⑥ 遍历每一列；语句 ⑦ 把 filter_data 对应的行向量赋值给输出 4D 特征图对应的深度维度。

需要特别强调的是，第 (3) 步运算和常规的神经网络是一致的，先进行矩阵相乘，然后进行非线性激活，只是由于输入和输出是 3D 特征图，需要对特征图进行形状的转换。

本书后面解释 CNN 结构时，把非线性激活层 ReLU 包含在卷积层里面，不单独作为一层。整个程序结构清晰，逻辑简单，可读性很强，程序运行效率高，希望读者仔细研读。我建议读者先理解 NumPy 中 array 多维数组的存储模式。array 中元素存储模式是先存储最后维度的数据，然后依次存储前一维度的数据，最后存储第一个维度数据。对于读者熟悉的 2D 数据 data = np.random.randn(h,w)，先存储第二维度的数据，也就是先存储行数据（每行数据有 w 个元素），一行

一行地依次存储。对于本程序中的 4D 特征图，读者很少用，但其存储模式和 2D 类似，`data = np.random.randn(b, h, w, d)`是以第四维度的数据（有 d 个元素）为存储单位的，依次存储这 d 个元素。如果这 d 个元素存储了 w 次，就相当于存储完第三维数据。如果这 d 个元素存储了 $w \times h$ 次，就相当于存储完第二维数据，等等。为了快速读取数据，数据存储地址最好连在一起，所以 array 数组最好是依次读取最后维度的数据，不要依次读取其他维度的数据。例如：

```
import time

h = 1000
w = 1000
weights = np.random.randn(h, w)

data = np.random.randn(w)
# 2.4ms
t1 = time.clock()
for i in range(h):
    data += weights[i,:]
t2 = time.clock()
print('run time(ms):', 1000*(t2 - t1) )

data = np.random.randn(h)
# 6.0ms
t1 = time.clock()
for i in range(w):
    data += weights[:,i]
t2 = time.clock()
print('run time(ms):', 1000*(t2 - t1) )
```

可以看出，两个程序块的数据吞吐量一样，速度却相差两倍以上，每次读取行的程序比读取列的程序快很多，因为行数据块的存储地址是连续的，而列数据块的地址是断续的。根据这个原理和卷积网络的局部连接性质，4D 数据的维度需定义为：`data = np.random.randn`

(batch, height, width, depth)。需要注意的是，最后维度是深度，第一维度是批量。程序依次读取深度维数据，即最后一个维度的数据，以达到加速目的。

4.3　池化层

通常，卷积层的超参数设置为：输出特征图的空间尺寸等于输入特征图的空间尺寸。这样如果卷积网络里面只有卷积层，特征图空间尺寸就永远不变。虽然卷积层的超参数数量与特征图空间尺寸无关，但这样会带来一些缺点。

(1) 空间尺寸不变，卷积层的运算量会一直很大，非常消耗资源。

(2) 卷积网络结构最后是通过全连接层输出分值向量的，如果空间尺寸一直不变，则全连接层的权重数量会非常巨大，导致过拟合。

(3) 前面几层的卷积层的输出存在大量冗余，如果空间尺寸不变，则冗余会一直存在，因此需要一种技巧来减小空间尺寸。

4.3.1　概述

池化是一种最常用的减小空间尺寸的技巧，它可以对输入的每一个特征图独立地降低其空间尺寸，而保持深度维度不变。首先对特征图的每个局部窗口数据进行融合，得到一个输出数据，然后采用大于 1 的步长扫描特征图。最常见的局部窗口尺寸是 2×2，有时也会采用 3×3，步长是 2 会去除 75% 的神经元，步长如果采用 3，则会去除 88.89%

的神经元，这过于剧烈，实践中不会采用。由于池化操作会去除大量的神经元，所以可以看作一种提纯操作。对局部窗口数据进行融合，最常使用的是 MAX 操作，即选取局部窗口数据的最大值。当然，也可以采用取平均值操作，但不常用。图 4.5 展示了池化层操作示意图，输入特征图尺寸 4×4×3 被降采样到了 2×2×3，采取的滤波器尺寸是 2，步长为 2。采用最大值池化，2×2 的局部区域选取最大值。

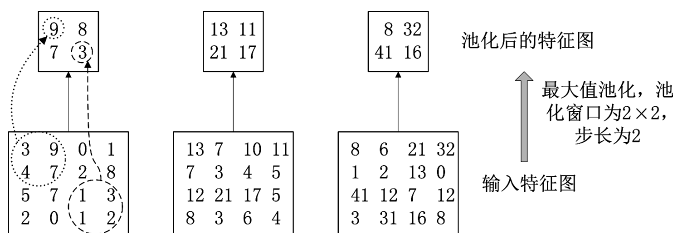


图 4.5　池化层操作示意图

为什么采用最大值进行池化操作？这是因为卷积层后接 ReLU 激活，ReLU 激活函数把负值都变为 0，正值不变，所以神经元的激活值越大，说明该神经元对输入局部窗口数据的反应越激烈，提取的特征越好。用最大值代表局部窗口的所有神经元，是很合理的。最大值操作还能保持图像的平移不变性，同时适应图像的微小变形和小角度旋转。

最后强调，池化只减小空间维度尺寸，深度维度的尺寸保持不变。如前所述，深度维度的尺寸可以看作空间位置处神经元提取的特征数量。随着空间尺寸的减小，神经元的感受野越来越大，即神经元观察到的局部区域越来越大，所以需要提取更多的特征，故深度维度一般会随着空间尺寸的减小而增大。

下面简要介绍池化层的一些参数。

- 输入特征图的尺寸为 $H1 \times W1 \times D1$
- 2 个超参数：
  - 滤波器的空间尺寸 F
  - 步长 S
- 输出特征图的尺寸为 $H2 \times W2 \times D2$，其中：
  - $H2 = (H1 - F)//S + 1$
  - $W2 = (W1 - F)//S + 1$
  - $D2 = D1$

对输入特征图进行固定的操作，所以没有可学习的参数；池化层中很少使用 0 填充。

不使用池化层：如前所述，池化层的目的是减小特征图的空间尺寸，卷积层也可以减小空间尺寸，即采用步长 $S = 2$ 来降低特征图的空间尺寸，卷积层与步长为 2 的池化操作一样，会去除 75% 的神经元。

4.3.2 池化层代码实现

池化层将每个局部窗口的数据转化为小矩阵，按行堆叠成大矩阵，然后每行取最大值得到大的列向量，最后转化为 3D 特征图。

(1) 输入 3D 特征图转化为矩阵 $\boldsymbol{X}$：局部区域转化为小矩阵。比如，输入是 $56 \times 56 \times 96$，局部窗口尺寸为 2×2，步长为 2 进行池化，取输入中的 $2 \times 2 \times 96$ 局部数据块，将其转化为尺寸为 96×4 的小矩阵，注意是将 96 维的深度向量拉伸为一个列向量，共有 4 个深度向量。以步

长为 2 扫描每一个局部窗口，所以输出的宽高均为 (56−2)//2+1 = 28，共有 28 × 28 = 784 个局部窗口，784 × 96 = 75 264 个行向量，输出矩阵 ***X*** 的尺寸是 75 264 × 4。因为局部窗口之间没有重叠，所以输入特征图中的元素在不同的行中没有重复。

(2) 最大值池化：提取大矩阵每行的最大值，`matric_data.max(axis = 1,keepdims = True)`，即得到每个局部窗口的最大值。在本例中，这个操作的输出是大列向量 [75 264 × 1]。

(3) 输出新 3D 特征图 28 × 28 × 96，大列向量的每 96 个元素构成输出 3D 特征图的一个深度向量。

由于池化操作简单直观，读者也有卷积层程序的基础，因此直接给出批量数据池化层的代码：

```
    filter_size = 2
    filter_size2 = filter_size*filter_size
    stride = 2

    (batch, in_height, in_width, in_depth) = (8, 32, 48, 16)
    in_data = np.random.randn(batch, in_height, in_width,
        in_depth)

    out_height = (in_height - filter_size)//stride + 1
    out_width = (in_width - filter_size)//stride + 1
    out_size = out_height*out_width
    out_depth = in_depth
    out_data = np.zeros((batch, out_height, out_width, out_depth))

①   matric_data = np.zeros( (out_size*batch*in_depth,
        filter_size2) )
```

```
    height_ef = in_height - filter_size + 1
    width_ef = in_width - filter_size + 1

    for i_batch in range(batch):
②       i_batch_size = i_batch*out_size*in_depth
        for i_h, i_height in zip(range(out_height), range(0,
            height_ef, stride)):
            i_height_size = i_batch_size + i_h*out_width*in_depth
③           for i_w, i_width in zip(range(0, out_width*in_depth,
                in_depth),
                    range(0, width_ef, stride)):
④               md = matric_data[i_height_size + i_w :
                    i_height_size + i_w + in_depth, : ]
⑤               src = in_data[i_batch, i_height : i_height +
                    filter_size,
                        i_width  : i_width  + filter_size, :]
                for i in range(filter_size):
                    for j in range(filter_size):
⑥                       md[:, i*filter_size + j] = src[i, j, :]

⑦   matric_data_max_value = matric_data.max(axis = 1,
        keepdims = True)
⑧   matric_data_max_pos = matric_data == matric_data_max_value

    for i_batch in range(batch):
        i_batch_size = i_batch*out_size*out_depth
        for i_height in range(out_height):
            i_height_size = i_batch_size +
                i_height*out_width*out_depth
            for i_width in range(out_width):
⑨               out_data[i_batch, i_height, i_width, :] =
                    matric_data_max_value[
                                i_height_size +
                                    i_width*out_depth :
                                i_height_size +
                                    i_width*out_depth +
                                    out_depth].ravel()
```

首先，设置超参数，随机生成输入 4D 特征图，然后计算输出特征图的维度，分配存储空间。语句①是分配大矩阵存储空间，注意行数量的计算公式，要乘以 `in_depth`；语句②是计算第 `i_batch` 个 3D 特征图的首行位置，注意要乘以 `in_depth`，并且后面所有计算行位置的代码都需乘以 `in_depth`；语句③遍历列，这里也要乘以 `in_depth`；语句④取出 `in_depth` 数量的行向量；语句⑤获得输入特征图的局部窗口数据；语句⑥赋值空间位置 (i, j) 的深度维度的数据；语句⑦执行最大值滤波，注意 `keepdims = True` 参数设置，保持矩阵的维度；语句⑧保存最大值位置，是为了计算梯度的需要；语句⑨赋值深度维度的数据。理解该程序的核心是语句⑥，它把空间位置 (i, j) 处的深度维度上的 `in_depth` 个数据作为一个深度向量，赋值给大矩阵的列。虽然池化层运算比卷积层简单，理解起来也容易，但程序实现比较复杂。

4.4　全连接层

如果卷积网络输入是 $224\times224\times3$ 的图像，经过一系列的卷积层和池化层（因为卷积层增加深度维度，池化层减小空间尺寸），尺寸变为 $7\times7\times512$，之后需要输出类别分值向量，计算损失函数。假设类别数量是 1000（ImageNet 是 1000 类），则分值向量可表示为特征图 $1\times1\times1000$。如何将 $7\times7\times512$ 的特征图转化为 $1\times1\times1000$ 的特征图呢？最常用的技巧是全连接方式，即输出 $1\times1\times1000$ 特征图的每个神经元（共 1000 个神经元）与输入的所有神经元连接，而不是局部连

接。每个神经元需要权重的数量为 $7\times7\times512=25\,088$，共有 1000 个神经元，所以全连接层的权重总数为：$25\,088\times1000=25\,088\,000$，参数如此之多，很容易造成过拟合，这是全连接方式的主要缺点。

全连接层的实现方式有两种。一种方式是把输入 3D 特征图拉伸为 1D 向量，然后采用常规神经网络的方法进行矩阵乘法；另一种方式是把全连接层转化成卷积层，这种方法更常用，尤其是在物体检测中。

4.4.1 全连接层转化成卷积层

全连接层和卷积层中的神经元都是计算点积和非线性激活，函数形式是一样的，唯一的差别在于卷积层中的神经元只与输入数据中的一个局部区域连接，并且采用参数共享；而全连接层中的神经元与输入数据中的全部区域都连接，并且参数各不相同。因此，两者是可能相互转化的。

- **卷积层转换为全连接层**：任意一个卷积层都能转换为等价的全连接层。此时连接整个输入空间的权重矩阵是一个巨大的矩阵，除了某些特定区域（局部连接）外，其余都是零。不同神经元的矩阵，其非零区域的元素都是相等的（参数共享），详见 4.2.4 节。
- **全连接层转化为卷积层**：比如，一个全连接层，输入特征图是 $7\times7\times512$，输出特征图是 $1\times1\times1000$，这个全连接层可以等效为一个卷积层：$F=7$，$P=0$，$S=1$，$K=1000$。即将卷积核的尺寸设置为和输入特征图的空间尺寸一致，不需要 0 填充，也不需要滑动卷积窗口，所以输出空间尺寸为 1，只有一个单

独的深度向量，所以输出变成 $1\times1\times1000$。全连接层转化为卷积层操作，还会带来额外的好处：可以在一次前向传播中，让卷积网络在一幅更大的输入图像中的不同位置进行卷积。

如果 $224\times224\times3$ 的输入图像经过多次卷积层和池化层之后得到特征图 $7\times7\times512$，即空间尺寸缩小了 32 倍。那么，$384\times384\times3$ 的大图像经过同样的卷积层和池化层之后，会得到特征图 $12\times12\times512$（$384/32=12$）。然后再经过一个全连接层转化得到的卷积层，最终输出为 $6\times6\times1000$（卷积 $F=7$，步长 $S=1$ 后空间尺寸为 $(12-7)//1+1=6$）。而 $(384-224)//32+1=6$，相当于采用尺寸为 $F=224$ 的卷积核，以步长 $S=32$，对 $384\times384\times3$ 的图像进行了 $6\times6=36$ 次卷积。

全连接层转换为卷积层后的卷积网络只需进行一次前向传播，就和 36 次卷积的效果是一样的。相比之下，一次前向传播计算要高效得多，因为共享了计算资源。这一技巧在实践中经常被使用，特别是在物体检测领域。通常，输入一张尺寸大的图像，使用变换后的卷积网络对空间上很多不同位置的子图像进行评估，得到分值向量，然后求这些分值向量的平均值。如上，得到 $6\times6\times1000$ 特征图后，再对每个 6×6 特征图进行平均,得到最终特征图 $1\times1\times1000$,效果一般会更好。

上述操作只能得到步长为 32 的卷积效果，如果想用步长小于 32，如 16，最终会得到 $11\times11\times1000$ 的输出，因为 $(384-224)//16+1=11$。这一问题可以采用两次前向传播解决，第一次在原始图像进行卷积，得到 $6\times6\times1000$ 的输出；第二次分别沿宽度和高度平移 16 个像素，得到“新图像”$368\times368\times3$，然后在“新图像”上进行卷积，得到

$5\times5\times1000$ 的输出，因为 $(368-224)//32+1=5$。两次结果合并，得到 $11\times11\times1000$。

4.4.2 全连接层代码实现

代码可以直接采用 4.2.6 节的卷积代码，只是超参数固定为 $S=1$，$P=0$，卷积核尺寸 F 一般等于输入特征图的空间尺寸，这样输出特征图空间尺寸为 1×1。如果卷积核尺寸小于空间尺寸，则输出特征图空间尺寸将大于 1×1。当卷积核尺寸 F 等于输入特征图的空间尺寸时，可以采用把输入 3D 特征图拉伸为 1D 数据的方式进行计算，此时代码如下：

```
    last = 0
    (batch, in_height, in_width, in_depth) = (8, 32, 48, 16)
    in_data = np.random.randn(batch, in_height, in_width,
       in_depth)

    size = in_height * in_width * in_depth
    matric_data = np.zeros( (batch, size) )

    for i_batch in range(batch):
①       matric_data[i_batch] = in_data[i_batch].ravel()

    out_depth = 32

    weights = 0.01 * np.random.randn(size, out_depth)
    bias = np.zeros((1, out_depth))

    filter_data = np.dot(matric_data, weights) + bias

    if not last:
②       out_data = np.maximum(0, filter_data) # ReLU 激活
```

语句 ① 把 3D 特征图拉伸为 1D 向量；语句 ② 进行非线性激活。注意，最后一层全连接层输出分值，不需要非线性激活。该程序除了多了拉伸语句，其他代码和常规神经网络代码一模一样。

4.5　卷积网络的结构

前面介绍了卷积网络的 3 种基本模块：卷积层（CONV）、池化层（POOL，默认最大值池化）和全连接层（FC）。卷积层和全连接层后面都需紧接 ReLU 激活层，为了简化书写，省略此层。但必须注意的是，最后一个全连接层后面不需接 ReLU 激活层。那么，如何通过这些模块组织成卷积网络呢？

4.5.1　层的组合模式

卷积网络最基本的结构是：先堆叠一个或多个卷积层进行特征提取，然后接一个池化层进行空间尺寸缩小，之后重复此模式，直到空间尺寸足够小（如 7×7 和 5×5），最后接多个全连接层，其中最后一个全连接层输出类别分值。

例如，下面是一些常见的网络结构。INPUT → [CONV → POOL] × 3 → FC → FC：每个卷积层之后紧跟一个池化层，重复了 3 次。INPUT → [CONV × 2 → POOL] × 3 → FC × 2 → FC：每 2 个卷积层之后有一个池化层，堆叠多个卷积层可以学习到更丰富的特征。INPUT → [CONV → POOL] → [CONV × 2 → POOL] × 2 → [CONV × 5 → POOL] × 2 → FC × 2 → FC：堆叠卷积层的数量随着网络加深而变大，如开始时候，一个

卷积层后立即池化，最后是 5 个卷积层才池化一次。因为随着特征图空间尺寸的减小，神经元的感受野越来越大，提取的特征更具有全局性，需要提取更复杂的关系，所以需要更多的卷积层。这种结构现在十分流行。

输入图像的空间尺寸常用的是 32（CIFAR-10）、64、96（STL-10）、224（ImageNet）、384 和 512，这些尺寸能被 2 整除很多次。

卷积层使用小尺寸卷积核（3×3），步长 $S = 1$，如果必须使用大的卷积核（5×5 或者 7×7），通常只用在第一个卷积层中，其输入是原始图像，且仅使用一次。

池化层对特征图进行空间降采样，最常用的是 2×2 感受野的最大值池化，步长为 2，另一个不常用的是 3×3 感受野，步长为 2。

为什么使用步长 $S = 1$ 的卷积层？实践表明小步长的效果更好。步长为 1 的卷积层不改变输入特征图的空间维度，只对深度维度进行变换。

为何卷积层使用零填充？因为对特征图进行零填充，不会改变输入特征图的空间尺寸，$P = (F - 1)//2$。如果不进行零填充，每次卷积后特征图的尺寸就会减小 2，那么特征图边缘的信息就会过快地损失掉，特别是堆叠很多个卷积层的时候。

为什么要堆叠多个卷积层，而不直接用一个卷积核尺寸大的卷积层？假设一层一层地堆叠了 3 个 3×3 的卷积层，第一个卷积层中的每个神经元对输入特征图有 3×3 的感受野，第二个卷积层上的神经元对

第一个卷积层有 3×3 的感受野，即对输入特征图有 5×5 的感受野，同样，第三个卷积层上的神经元对第二个卷积层有 3×3 的感受野，即对输入特征图有 7×7 的感受野。假设不采用堆叠 3 个 3×3 的卷积层，而是使用一个单独的 7×7 的感受野的卷积层，那么所有神经元的感受野也是 7×7，但这样有一些缺点：首先，多个卷积层与非线性激活层的交替结构，比单一的卷积层结构提取的特征更富有表现力；其次，假设输入输出的特征图的深度维度都是 C，那么单独的 7×7 卷积层包含 $C\times(7\times7)\times C=49C^2$ 个权重，而 3 个 3×3 的卷积层只有 $3\times(C\times(3\times3)\times C)=27C^2$ 个权重。堆叠多个 3×3 的卷积层缺点是：在进行误差反向传播时,需要存储中间每个卷积层的激活值,这会占用更多的内存。

为什么卷积网络称为深度网络？这里的深度是指网络的深度，一般来说，网络隐含层大于两层，就能称为深度网络，也就是深度学习，卷积网络的深度一般都大于 5 层，甚至达到上千层！传统的机器学习方法可以看作浅层网络，如 SVM 可以看作一个隐含层的网络，深度学习之前的神经网络隐含层一般不超过两层。卷积网络的深度主要是由图像的多层次结构决定的：网络的前层学习图像的低层特征，如边缘和纹理模式；中间层学习图像的中层视觉特征，如眼睛、腿等物体部件；后层网络学习物体整体概念，最后的全连接层得到物体类别。

卷积网络最基本的结构是将卷积层、池化层和全连接层这 3 层进行简单的串联（VGG 为其代表），虽然结构清晰，容易掌握，但网络学习效率不高。谷歌的 Inception 结构和微软亚洲研究院的残差（Residual Net）结构，虽然连接模式复杂，但网络参数更少，学习效率更高。这三种最经典的结构，本书后面都有论述。

4.5.2 表示学习

可以从另一个角度来理解上述卷积神经网络。卷积网络的输入是图像，如 $224\times224\times3$，经过多个卷积层和池化层后，假设输出为 $7\times7\times512$，最后接多个全连接层输出分值向量。多个全连接层可以看作常规神经网络，该网络的输入是 $7\times7\times512$。这样可以把卷积网络全连接层前的多个卷积层和池化层看作特征提取器：每层对上一层的输出进行特征变换，把与类别无关的低层表示 $224\times224\times3$（如图像）变换为与类别密切相关的高层表示 $7\times7\times512$，使得原来基于全连接层难以完成的任务成为可能。每层一般是对 3D 特征图的空间尺寸进行减小，深度尺寸进行增加，即从 $224\times224\times3$ 最终变换为 $7\times7\times512$。所以卷积网络也称为“特征学习”或“表示学习”。

深度学习之前，机器学习进行模式分类，这个过程需要专家提取特征，这称为“特征工程”，见第 1 章内容。特征的好坏对模型的泛化性能至关重要，专家设计出好特征十分困难，特别是在图像和声音领域，图像领域著名的人工特征有 SIFT、HOG 和 LBP 等。卷积网络通过机器学习技术，自己从数据中产生好特征，把人类从特征工程中解放出来，大大扩展了机器学习的适用领域。

机器学习技术自身产生的好特征是指针对特定的训练集来说的，对其他训练集可能不是。如果该特征对很多不同的训练集都是好特征，就说明该模型具有很强的迁移学习能力，是十分理想的模型。举个例子，训练卷积网络对于文字的识别能力，如果训练集里面的字体都是宋体，网络对训练集取得很好的分类效果，这时就可以认为模型学习

出了好特征。但是该模型对于行书字体的识别效果很差，说明该模型学习到的特征对于行书字体而言，不是好特征。

4.6　卷积网络的神经科学基础

生物学启发人工智能最为成功的案例可能就是卷积网络，卷积网络的一些关键设计原则均来自神经科学。神经生理学家 David Hubel 和 Torsten Wiesel 通过对猫的视觉系统的多年研究发现：初级视觉皮层 V1 细胞具有强烈的方向选择性，即对视野中特定方向的条纹反应强烈，而对其他方向的条纹几乎没有反应。初级视觉皮层 V1 是大脑对视觉输入开始执行显著高级处理的第一个区域，卷积网络层的设计借鉴了 V1 的 3 个性质。V1 用二维结构来反映视网膜的图像结构。卷积网络通过用二维映射定义特征的方式来描述该特征，如 3D 特征图。V1 包含许多简单细胞，这些细胞的活动在某种程度上可以概括为：在一个小的空间位置接受域内的图像的线性函数。卷积网络的神经元设计为局部连接的卷积运算。V1 还包含许多复杂细胞，这些细胞响应类似于简单细胞检测的那些特征，但是复杂细胞对于特定的位置的微小偏移具有不变性，这启发了卷积网络的池化单元。

大多数的 V1 细胞具有由 Gabor 函数所描述的权重，而 Gabor 函数是高斯函数和余弦函数的乘积。卷积网络第一层卷积层学习到的特征和 Gabor 函数非常类似。

可惜的是，神经科学很难告诉我们该如何训练卷积网络。

第二部分

优化篇

第 5 章

基于梯度下降法的最优化方法

前面几章详细介绍了参数化的特征映射模型：线性模型、神经网络和卷积神经网络。我们通常先通过这些模型得到分值向量，再通过损失函数来评价模型参数的优劣。那么，这些参数如何取到最优值呢？目前的主流方法是基于梯度下降法的各种改进算法。第 2 章已经介绍了梯度下降法，本章将详细介绍各种改进算法。

梯度下降法的基本思想：如果要找到某函数的最小值，最好的方法是沿着负梯度方向探寻。假设有一个参数向量 $\boldsymbol{x}$ 及其梯度 $\mathrm{d}\boldsymbol{x}$，更新形式是：

$$\boldsymbol{x} \mathrel{+}= -\mathrm{lr} \times \mathrm{d}\boldsymbol{x}$$

其中 lr 称为学习率，是超参数，是正的小常量。在整个数据集上计算梯度进行参数更新时，只要学习率足够小，每次更新参数时总能使损失函数值减小。参数初始化相关的内容可参见 5.9 节，梯度计算方法见第 6 章。

5.1　随机梯度下降法 SGD

参数更新中的梯度是通过对训练集中所有样本的平均损失求梯度得到的。当训练集规模较小时，计算梯度不成问题。然而对于大规模数据集（比如 ImageNet），当训练样本达到百万量级时，需要计算整个训练集的梯度，才能更新一次参数。这样更新效率太低，浪费计算资源。一个常用的代替方法是从训练集中随机抽取小批量样本，计算它们平均损失的梯度，来实现一次参数更新。小批量样本可包含 64 个或 128 个样本，而整个训练集有 120 万个样本。

用小批量样本的平均损失代替全体样本的平均损失进行参数更新，可以加快参数更新频率，加速收敛。小批量样本的平均损失是全体样本平均损失的无偏估计。这是因为训练集中的同类样本是相关的，同类样本中不同个体的损失是相似的，所以随机抽取的一个样本损失可以作为该类所有样本损失的估计。

如果小批量样本中只有一个样本，那么称为随机梯度下降法（Stochastic Gradient Descent，SGD）。SGD 使用不广，因为矩阵的向量化操作使一次计算 128 个样本的梯度比 128 次计算一个样本的梯度要高效很多。SGD 指每次使用一个样本来更新参数，但我们经常使用 SGD 来指代小批量梯度下降法。小批量样本的数量是一个超参数，它受存储器的存储容量限制，一般设置为 32、64、128 等 2 的指数。这是因为许多矩阵向量化操作实现时，如果数据量是 2 的指数，运算效率会更高。SGD 及其改进算法是深度学习中最常用的优化算法，本章主要介绍这些算法。

SGD 算法中，每次都要随机抽取 batch 个样本，实现时可以采用先整体打乱训练集，然后每次按顺序取 batch 个样本的方式。这种实现方法效率高，代码如下：

```
def calu_gradient(batch_data, labels, x):
    return np.random.randn(np.size(x))

num_train_samples = 1000
im_height = 32
im_width = 32
im_dims = 3
num_class = 10
batch = 20
lr = 10**(-4)
dim_x = 1000
x = np.random.randn(dim_x)

train_data = np.random.randn(num_train_samples, im_height,
    im_width, im_dims)
train_labels = np.random.randint(num_class,
    size = num_train_samples)

epoch_num = 20
for epoch in range(epoch_num):
    shuffle_no = list(range(num_train_samples))
①   np.random.shuffle(shuffle_no)

②   train_labels = train_labels[shuffle_no]
    train_data = train_data[shuffle_no]

    for i in range(0, num_train_samples, batch):
③       batch_data = train_data[i:i+batch,:]
        labels = train_labels[i:i+batch]

④       dx = calu_gradient(batch_data, labels, x)
⑤       x += -learning_rate*dx
```

首先定义梯度计算函数，这里采用随机数模拟梯度，具体计算见第 6 章。定义一些相关变量和学习率，并随机生成训练集。语句 ① 获得随机打乱的整数序列，语句 ② 整体打乱训练集，语句 ③ 顺序取出 batch 个样本，语句 ④ 计算梯度，语句 ⑤ 更新参数。所有训练样本都训练一次，称为一个周期（epoch）。注意，每个训练周期开始时，必须重新打乱训练集。

基于 SGD 的改进算法，只有语句⑤的参数更新不同，前面程序都是一样的，故后面只给出更新参数的程序段。

5.2　基本动量法

把梯度下降法想象成小球从山坡滚向山谷的过程，损失值是小球当前的高度，最小化损失值就是希望小球滚动到高度最小的山谷。设设小球的空间位置是 x，每次移动的路径矢量是 dx，位置 x 处的山的高度就是损失值。那么基本梯度下降法的小球是这样“滚动”的：在出发点 A 处，计算点 A 在各个方向的坡度，沿着坡度最陡的方向移动一段距离到达 B 点，然后停下来。在 B 点再计算坡度最陡的方向，沿着这个方向走一段路，再停下。注意每到一个新位置，小球都必须停下来。准确地说，小球并没有滚动下山，而是盲人下山的方式，走一步停一步，用拐杖探明坡度最陡的方向，沿此方向移动一步后停下来，周而复始，直到到达山谷或平地，此时各个方向上的坡度都为 0，盲人无法移动了。一个真正的小球要比盲人高效得多，从起始点 A 静止滚动到点 B 的时候，小球获得一定速度，继续滚动，小球会越滚越快，

快速滚向谷底。动量法就是通过模拟小球的滚动过程来加速神经网络的收敛。这时就需要一个速度变量，根据速度变量来更新参数。速度变量积累了历史梯度信息，使之具有惯性，当梯度方向一致时，加速收敛；当梯度方向不一致时，减小路径曲折程度。代码如下：

```
    mu = 0.9
①   v = mu*v
②   v += - lr*dx
    x += v
```

变量 v 就是速度，初始化为 0（相当于小球从静止开始滚动），语句 ① 表明 v 累加了历史梯度，mu 是小于 1 的正超参数，物理意义类似于摩擦系数，该变量有效地抑制了速度，降低了小球的动能，不然小球在山谷永远不会停下来（如果没有摩擦，滚动的小球会一直运动下去，不会停在山谷）。通过交叉验证，这个参数通常设为 0.5、0.9 或 0.99。但需注意的是：mu 越大，摩擦越小；mu = 1，没有摩擦；mu = 0，摩擦无穷大，变为基本的梯度下降法。语句 ② 表明速度 v 受当前梯度 dx 调节。

假设每个时刻的梯度 dx 相等，由语句 ① 和语句 ② 可得：$v = -\mathrm{lr}/(1-\mathrm{mu}) \times \mathrm{d}x$，此时相当于学习率为 lr/(1 − mu)，mu = 0.9 表示 10 倍于 SGD 算法的收敛速度。把 1/(1 − mu) 看作放大率更容易理解，mu 越大，放大率越大，收敛可能越快。

假设每个时刻的梯度 dx 总是 0（相当于小球滚动到平地），由语句 ① 可得：$v = \mathrm{mu}^n \times v_0$，其中 n 是参数更新次数。可见，v 是指数衰减，小球只要滚动到平地时的初速度 v_0 足够大，就有机会冲出当前平

地，到达一个更低的山谷。由于历史原因，速度 v 被称为动量，所以该方法称为动量法。

5.3　Nesterov 动量法

基本动量法每步的前进方向都是由下降方向的历史累积 v 和当前点的梯度方向 dx 合成的。换个角度看，可以认为下降方向本来应该为 v（语句 ②），但需要考虑当前位置处的实际情况来调整 v，调整量就是当前梯度 dx（语句 ③）。上述方法是由当前信息调整 v，我们能不能通过 v 的历史信息进行加速呢？由于每次更新时，v 也会发生改变，$dv = v - v_{pre}$，即用前后两次之差 dv 来调整 x，达到加速收敛，其中 v_{pre} 是上次的速度（参见语句 ①）。这种方法被命名为 Nesterov Accelerated Gradient，简称 NAG。其代码如下：

```
    mu = 0.9
①  pre_v = v
②  v = mu*v
③  v += - lr*dx
④  x += v + mu*(v - pre_v)
```

5.4　AdaGrad

前面讨论的 3 种方法都是对参数向量进行整体操作，对向量每个元素的调整都是一样的。所有元素要同时调整到一个较优的学习率是很困难的，需要经过反复测试，耗费大量的计算资源，因此自适应地调整学习率，甚至是逐参数的自适应调整学习率成为了当下研究的热

点。这些方法一般会引入新的超参数，但是训练集对这些超参数不敏感，参数设置比较容易。下面介绍一些在实践中可能会遇到的常用自适应算法。首先是 AdaGrad：

```
cache += dx**2
x += - (lr / (np.sqrt(cache) + eps)) *dx
```

注意，变量 cache 初始化为 0，它和梯度向量 dx 同维度，每个元素累加了对应梯度元素的历史平方和。cache 用来归一化参数以更新步长，归一化是逐元素进行的（lr / (np.sqrt(cache) + eps)）。所以，对于高梯度值的权重，其历史累积和大，等效学习率减小，更新强度减弱；对于低梯度值的权重，其历史累积和小，等效学习率增加，更新强度增强。有意思的是，cache 开平方根非常重要。如果直接使用 cache，算法的效果会差很多。除数加小常量 eps（一般设为 1e－4 到 1e－8 之间）可以防止除数为 0。

假设梯度向量 dx 的某个元素 da 一直保持不变，则可得该元素的更新规律为：a +=－lr/np.sqrt(n)×sign(da)。可见，其更新量与梯度大小无关，更新方向与梯度方向相反。更新量与梯度大小无关，因此小的梯度元素也能达到较快的更新速率，这是 AdaGrad 的主要优点。更新量随着迭代次数 n 的增加而减小，所以 AdaGrad 的一个缺点是，从训练开始就累积梯度平方和会使有效学习率过早、过量地减小，这会导致过早停止学习。

第一步迭代（$n=1$）时，任意一个参数 a 的更新公式为：a +=－lr×sign(da)。改变量为固定值 lr，结合 5.9 节的参数初始化可知，a 初始化为正态分布，所以 lr 的取值不能太大，不能大于初始化的标准差，

否则参数的随机正态分布就失去意义，变成固定值初始化。

当 dx 为 0 时，更新停止，即 5.2 节中的小球不可能冲出当前平地，即鞍点。

5.5 RMSProp

RMSProp 方法采用指数衰减的方式，让 cache 丢弃遥远过去的历史梯度信息，只对最近的历史梯度信息进行累加，使之不那么激进，单调地降低学习率。其修改方式很简单，使用一个梯度平方的指数加权的移动平均：

```
decay_rate = 0.9
cache = decay_rate * cache + (1 - decay_rate) * (dx**2)
x += - (lr / (np.sqrt(cache) + eps)) *dx
```

变量 cache 初始化为 0，decay_rate 是一个超参数，常用的值是 {0.9, 0.99, 0.999}，x 更新和 AdaGrad 是一样的，只是 cache 变量不同，看起来更像动量法的 v。RMSProp 和 AdaGrad 一样，仍然是基于梯度的历史累加和来对每个权重的学习率进行调整。不同的是，其更新主要使用最近的梯度信息，这不会让学习率一直单调变小。

假设梯度向量 dx 的某个元素 da 一直不变，则可得该元素稳定后的更新规律为：a += −lr × sign(da)。可见，其更新量是固定值 lr，更新方向与 da 相反，克服了 AdaGrad 的更新量随着 n 变大越来越小的缺点。

第一步迭代（$n=1$）时，任一元素 a 的更新公式为：a + = −lr/np.sqrt

(1−decay_rate) × sign(da)，改变量为固定值 lr/np.sqrt(1−decay_rate)。这与 AdaGrad 相比，改变量放大了 1/np.sqrt(1−decay_rate) 倍。

当 dx 为 0 时，更新停止，即 5.2 节中的小球不可能冲出当前平地。

5.6 Adam

Adam 看起来像是 RMSProp 的动量版，其更新方式如下：

```
    mu = 0.9
    decay_rate = 0.999
    eps = 1e-8
    v = mu*v + (1-mu)*dx
①  vt = v/(1 - mu**t)
②  cache = decay_rate *cache + (1-decay_rate )*(dx**2)
    cachet = cache/(1 - decay_rate**t)
    x += - (lr/ (np.sqrt(cachet) + eps)) * vt
```

其更新方法和 RMSProp 一样，但采用平滑版的动量 `v`，而不是原始梯度 `dx`，`v` 和 `cache` 初始为 0。语句 ① 中 `t` 是迭代次数，因为在完全热身之前存在偏差，需要采取一些补偿措施，使 `v` 和 `cache` 在刚开始训练时变大。训练次数 `t` 增大时，`mu**t` 趋近 0，语句 ① 在训练后期没有影响。

假设梯度向量 d$\boldsymbol{x}$ 的某个元素 da 一直不变，则可得该元素稳定后的更新规律为：a + = −lr × sign(da)。可见，其更新量是固定值 lr，更新方向与梯度方向相反。

第一步迭代（$n=1$）时，任一元素 a 的更新公式为：a += −lr × sign (da)。改变量为固定值 lr，同 AdaGrad 一样，lr 的取值不能太大，其默

认值是 `10**(-3)`。

当 dx 为 0 时，如同动量法，动量 v 是指数衰减的，有可能冲出平地。

5.7　AmsGrad

Adam 存在可能不收敛的缺点，因为其有效学习率为 lr/(np.sqrt(cache) + eps)，其中 cache 主要受最近的梯度历史信息影响，故其值波动较大。当它取比较小的值时，会使有效学习率很大，使之不能收敛。RMSProp 也有类似缺点。改进方法就是使有效学习率不能增加，只需在 Adam 方法中进行很小的修改，语句 ② 改为：

```
cache = np.max((cache, decay_rate *cache +
    (1-decay_rate)*(dx**2)))
```

这样 cache 随着迭代次数的增加而变化时，一定不会减小，这样有效学习率就不会增加，保证了收敛。

关于这 7 种优化方法，如何选择呢？推荐首先使用 Adam 或者 Nesterov 动量法，也可以尝试 AmsGrad。

5.8　学习率退火

训练深度网络的时候，学习率如果一直固定不变的话，损失函数就难以达到深谷。通常随着训练次数的增加，学习率会逐渐变小，这

就是学习率退火。以小球滚动为例，学习率很高时，小球的动能很大，小球会无规律地四处滚动，难以滚入损失函数更深、更窄的山谷里面。学习率很小时，则小球动能太小，滚动过慢。所以刚开始的时候，学习率应该取比较大的值，然后逐渐减小。速率减小得不能过慢，也不能过快。通常，学习率退火有如下 3 种方式。

- **随训练周期衰减**：每进行几个周期（epoch，所有样本都训练了一次称为一个周期），降低一次学习率。典型的做法是每 5 个周期学习率减少一半，或者每 20 个周期减少到 0.1，这些数值都是经验值。在实践中更为常见的做法是：每当验证集错误率停止下降时，就把学习率降低到原来的 1/10。
- **指数衰减**：数学公式是 $\alpha = \alpha_0 e^{-kt}$，其中 α_0 和 k 是超参数，t 是迭代次数。
- **反比衰减**：数学公式是 $\alpha = \alpha_0/(1+kt)$，其中 α_0 和 k 是超参数，t 是迭代次数。

实践中，最常用的退火方法是随训练周期衰减，因为其超参数（衰减比例和训练周期）具有更好的实际意义。最后，如果有足够的计算资源，可以让衰减更慢些，这样训练时间会更长些。

5.9 参数初始化

开始训练网络之前，需要初始化网络的权重。由于此时对任务没有先验知识，所以不可能知道某个属性对分类是正相关还是负相关，所以权重初始化为 0 比较合理。但是，全部权重都初始化为 0 是错误

的，那样会使网络中同一层的每个神经元都计算出同样的输出 0，然后它们就会在反向传播中计算出同样的梯度，从而进行同样的参数更新，这样神经元之间就失去了不对称性的源头。

小随机数初始化：权重初始值接近 0 但不能等于 0，是比较合理的。采用小的正态分布的随机数 $W = 0.01 \times \text{np.random.randn}(D, H)$ 来打破对称性。但并不是小数值一定会得到好的结果，这样会减慢收敛速度。因为在梯度反向传播的时候，会计算出非常小的梯度。

使用 1/sqrt(n) 来校准方差，使输出神经元的方差为 1，这样参数初始化为 $w = \text{np.random.randn}(n) / \text{sqrt}(n)$，其中 n 是神经元连接的输入神经元数量。实践证明，这可以提高收敛速度。

偏置初始化：通常将偏置初始化为 0 或者小常数（0.01），这是因为随机权重矩阵已经打破了网络对称性。

当前推荐的初始化为：当使用 ReLU 时，用 $w = \text{np.random.randn}(n) \times \text{sqrt}(2.0/n)$来进行权重初始化。代码如下：

```
in_depth = 128
out_depth = 32
std = np.sqrt(2/in_depth)
weights = std * np.random.randn(in_depth, out_depth)
bias = np.zeros((1, out_depth))
```

5.10　超参数调优

机器学习的各种模型，包括线性模型、常规神经网络和卷积网络

等，以及各种优化算法（如梯度下降法），都会引入很多超参数。贯穿本书的最重要的超参数有两个——初始学习率和正则化系数（L2 惩罚），还有很多值比较稳定的超参数，比如动量法中的动量参数 mu（一般取 0.9）等。如何确定最优的超参数？超参数不能通过学习算法进行优化，是人为指定的，所以为了获得最优的超参数，需要不断尝试不同取值，即在一定范围内搜索最优值。下面是一些注意事项。

首先，要设置合理的超参数范围，其学习率和正则化系数要在对数尺度上进行搜索。例如，一个典型的学习率搜索范围是：`lr = 10 ** uniform(-6, 1)`。这是因为通过学习率乘以梯度、正则化系数乘以权重，来对模型产生影响。但是有一些参数（`dropout`）还是在原始尺度上进行搜索（`dropout = uniform(0,1)`）。

其次，随机搜索优于网格搜索，它可以更精确地发现那些比较重要的好的超参数数值，而且程序也更容易实现。超参数随机搜索的代码如下：

```
①   lr = [1, -6]
②   reg = [-3, -6]
③   num_try = 10
    minlr = min(lr)
    maxlr = max(lr)
④   randn = np.random.rand(num_try*2)
⑤   lr_array = 10**(minlr + (maxlr - minlr)*randn[0: num_try])

    minreg = min(reg)
    maxreg = max(reg)
    reg_array = 10**(minreg + (maxreg - minreg)*randn[num_try:
        2*num_try])
    lr_regs =  zip(lr_array, reg_array)
```

```
⑥   for lr_reg in lr_regs:
        # train()
        pass
```

语句 ① 定义学习率对数搜索范围，语句 ② 定义正则化系数对数搜索范围，语句 ③ 定义随机搜索次数，语句 ④ 生成[0, 1]均匀随机数，语句 ⑤ 生成对数尺度的学习率数组，语句 ⑥ 对每一个超参数组合进行训练。

为了快速找到最优超参数，要先粗后精地分阶段搜索。实践中，先进行粗略范围（比如 `10 ** uniform(-6, 1)`）搜索，然后根据好结果出现的地方缩小搜索范围。进行粗搜索的时候，模型训练一个周期就可以了。因为如果超参数设定不合理，模型会无法学习，或者损失值会突然增大很多。在精细搜索阶段，要缩小搜索范围，模型需要运行多个周期，比如 5 个。最后，在最终的范围内进行仔细搜索，运行更多个周期，比如 20 个。

特别要注意边界上的最优值，一旦最优值位于边界上，则需重新设定包含该最优值的搜索范围，再次进行精细搜索。

由于超参数调优需要大量的计算资源，可能需要几天甚至几周、几个月，所以实践中只使用一个验证集，不进行交叉验证。同时需要监控每个训练周期后验证集的准确率和存储模型的记录点（记录点中应包含各种模型性能统计数据，比如损失值随训练周期的变化和验证集的准确率等），所以文件名中最好包含验证集的性能参数，方便后期查找和排序。

第 6 章

梯度反向传播算法

本章的核心问题是：给定参数化模型的损失函数 $f(x,w)$，其中 x 是输入数据，w 是模型参数，需要计算损失函数关于参数 w 的梯度。有了梯度,我们就能使用梯度下降法的各种改进方法进行模型优化了。注意损失函数输出的是标量，是一个数值，而不是向量或矩阵。

6.1　基本函数的梯度

神经网络模型包含最基本的三种函数：乘法、加法和非线性激活 ReLU。这三种函数的梯度计算都非常简单，列举如下：

$$
\begin{aligned}
&f(x,w)=xw\rightarrow\frac{\partial f}{\partial x}=w,\ \frac{\partial f}{\partial w}=x\\
&f(x,y)=x+y\rightarrow\frac{\partial f}{\partial x}=1,\ \frac{\partial f}{\partial y}=1\\
&f(x)=\max(0,x)\rightarrow\frac{\mathrm{d}f}{\mathrm{d}x}=I(x>0)
\end{aligned}
\tag{6.1}
$$

这里所有的变量都是标量，为一个数值。$I(x)$是指示函数，当括号内的

表达式为真时输出 1，否则输出 0。注意 $\max(0, x)$函数，在 0 点导数不连续，但是右导数为 1，左导数为 0，存在次导数。读者需要牢记偏导数的意义为函数值对于该变量的敏感程度，偏导数越大说明越敏感。偏导数为正，表示变量增加，函数值增加，是正相关；偏导数为负，表示变量增加，函数值减小，是负相关。

6.2　链式法则

对于复合函数，求梯度需采用链式法则。链式法则的内容如下：

$$z = f(y)，y = g(x)$$

则 $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$，即 z 对变量 x 的梯度是 z 对中间变量 y 的梯度与 y 对变量 x 的梯度的乘积。

常规神经网络和卷积网络都是多层网络，多层网络从函数角度看，就是复合函数。举个例子，为了突出链式法则，我们对模型进行简化，忽略偏置，令神经网络的每层神经元数量都为 1，这样都是标量，简化了偏导数求解，当两层的常规神经网络用于回归任务时，其数学模型为：

$$a_1 = xw_1,\ h_1 = \max(0, a_1),\ a_2 = h_1 w_2,\ \text{loss} = \frac{1}{2}(a_2 - y_0)^2 \tag{6.2}$$

其中 x 是输入，y_0 是 x 对应的标签，即需拟合的真实值。为了求 loss 对参数 w_1 的梯度，必须采用链式法则，先求每个函数的梯度，然后所

有梯度连乘，得到最终梯度：

$$\frac{\partial \text{loss}}{\partial a_2}=a_2-y_0,\ \frac{\partial a_2}{\partial h_1}=w_2,\ \frac{\partial h_1}{\partial a_1}=I(a_1>0),\ \frac{\partial a_1}{\partial w_1}=x \tag{6.3}$$

$$\frac{\partial \text{loss}}{\partial w_1}=(a_2-y_0)w_2I(a_1>0)x \tag{6.4}$$

这个例子表明，利用链式法则求梯度，只要能求得每个中间函数的梯度，则最终的梯度就是每个梯度的乘积。中间函数就是每层的变换函数，它们都由三种基本函数组成，求解十分简单，所以不存在困难。

这是一个稍微复杂的例子，函数形式：

$$f=\frac{x+y}{2x+y} \tag{6.5}$$

求 f 对 x 的偏导数。读者可以采用高数的方法直接求偏导数，这里为了清晰地展示链式法则和理解深度网络，我们采用分步法。分步法的关键是一个复杂函数通过中间变量，使之变成简单模块的嵌套，每个简单模块能直接利用公式得到偏导数，每个模块相当于一层网络。具体如下，先通过一些中间变量来计算函数值：

$$\text{num}=x+y,\ \text{den}=2x+y,\ \text{invden}=1/\text{den},\ f=\text{num}\times\text{invdena} \tag{6.6}$$

再对每个函数求偏导数，得：

$$\begin{array}{lll} \dfrac{\partial f}{\partial \text{num}}=\text{invden} & & \dfrac{\partial \text{den}}{\partial x}=2 \quad \dfrac{\partial \text{num}}{\partial x}=1 \\ & \dfrac{\partial \text{invden}}{\partial \text{den}}=-\text{invden}^2 & \\ \dfrac{\partial f}{\partial \text{invden}}=\text{num} & & \dfrac{\partial \text{den}}{\partial y}=1 \quad \dfrac{\partial \text{num}}{\partial y}=1 \end{array} \tag{6.7}$$

这里有两个路径可以得到 f 对 x 的偏导数：

$$
\begin{aligned}
\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial \text{invden}} \frac{\partial \text{invden}}{\partial \text{den}} \frac{\partial \text{den}}{\partial x} = \text{num} \times (-\text{invden}^2) \times 2 \\
\frac{\partial f}{\partial x} &= \frac{\partial f}{\partial \text{num}} \frac{\partial \text{num}}{\partial x} = \text{invden} \times 1
\end{aligned}
\tag{6.8}
$$

一定要注意，最终结果是这两个偏导数之和。

6.3　深度网络的误差反向传播算法

前面利用两层网络的模型来求损失函数对参数的偏导，该方法虽然能得到一些有意义的结果，但不能深入理解深度网络的梯度反向传播规律。本节采用 n 层网络模型，求损失函数对参数的偏导，达到深刻理解梯度反向传播规律的目的,这对理解深度网络训练特别有帮助。

同理，本节为了突出梯度反向传播规律，简化数学处理，令深度神经网络的每层神经元数量都为 1，这样都是标量，简化了求偏导数。神经网络用于回归任务，设 x 是输入，y_0 是 x 对应的标签，即需拟合的真实值，h_n 是网络的预测值，$f(x)$ 是非线性激活函数。令 h_0 等于 x，$f'(a_n)$ 等于 1，则数学模型为：

$$
\begin{aligned}
& a_1 = x w_1, h_1 = f(a_1) \\
& \quad \vdots \\
& a_i = h_{i-1} w_i, h_i = f(a_i) \\
& \quad \vdots \\
& a_n = h_{n-1} w_n, h_n = a_n \\
& \text{loss} = \frac{1}{2}(h_n - y_0)^2
\end{aligned}
\tag{6.9}
$$

首先，对每层求偏导数，然后进行连乘，就能得到损失 loss 对任一层参数 w_i 的梯度：

$$\frac{\partial \text{loss}}{\partial w_i} = (h_n - y_0)\prod_{j=i+1}^{n} w_j \prod_{j=i}^{n} f'(a_j) h_{i-1} \tag{6.10}$$

仔细分析式(6.10)，这里有几点请读者注意。

- $h_n - y_0$ 是预测值和真实值之间的误差，乘积中的变量是从最后一层开始向前递推的，所以该方法称为误差反向传播算法。有误差，就有梯度，就能学习，直到误差为 0，学习才结束。
- 表达式中有中间变量 a_j，所以为了加快梯度计算，需要在网络的前向计算过程中存储中间变量，但这样会占用大量内存。
- 表达式中含有权重的乘积，所以如果权重初始化得过小，梯度就会很小，导致学习速度慢。因为权重的绝对值小于 1，所以相乘的项越多，乘积越小，这导致越是靠前的层（i 较小）的参数越难学习。
- 表达式中有非线性激活函数导数的乘积。当激活函数采用 sigmoid 或 tanh 等函数时，由于其导数绝对值均小于 1，且存在饱和（导数趋近 0），这样乘积会很小。特别是对于网络前面的层，乘积会更小。这种现象在浅层网络不明显，但在深度网络中特别明显，因为层数 n 基本都在 10 以上，甚至成百上千。这样连乘的项很多，乘积会十分小（$0.5^{20} = 10^{-6}$），梯度太小会导致参数更新率低，学习困难。

- 如果激活函数是 ReLU，ReLU 的导数为 $I(x>0)$，即变量小于等于 0，导数为 0；变量大于 0，导数为 1。当网络采用合适的参数进行初始化和数据归一化时，可假设网络各层中间变量 a_j 有一半的概率小于 0，一半的概率大于 0，这样 loss 对参数 w_i 的梯度为 0 的概率十分高，并且随着 i 的减小会越来越高。当梯度为 0 时，根据梯度下降法，参数将无法更新，不能进行学习。在深度卷积网络中，损失函数对前面几层参数的梯度包含大量激活函数梯度的连乘，只要有一层激活函数的梯度为 0，则整个表达式就为 0。因此，网络中大量参数的梯度会为 0，易导致“ReLU 死亡”问题，所以前面几层的参数很难进行有效学习。
- 如果激活函数是 ReLU，虽然大部分非线性激活函数导数的乘积是 0，但还是有部分乘积不是 0，则这些乘积就是 1（虽然所占比例不高），不会像 sigmoid 函数那样，乘积都变得很小。乘积为 1 时，学习就比较容易，所以 ReLU 的收敛速度比 sigmoid 快不少，这也是 ReLU 大量使用的原因。

6.4　矩阵化

实际的网络中，所有的变量（包括输入、中间变量和参数）都是矩阵，这样求中间函数的梯度就是求矩阵对矩阵的梯度，非常困难。在上面的例子中，由于变量都是标量，所以中间函数的梯度也是标量，十分简单。这里通过一个数学技巧，避免直接计算中间函数的梯度，达到简化计算的目的，而且更便于理解。

因为损失函数是标量,最终的目的是计算损失函数对参数的梯度,且标量对矩阵参数的梯度是矩阵，所以梯度矩阵的尺寸和矩阵参数的尺寸一致。利用这个性质，能极大地简化梯度计算。任何机器学习模型的损失函数值都是标量，这一定要牢记在心，所以直接求损失函数对各层参数的梯度会十分方便。

这里直接给出结论：如果读者对矩阵求导感兴趣，可以去查看相关矩阵论的图书。

网络最后一层输出分值向量（向量是矩阵的特例）$\boldsymbol{S}$。假设我们得到损失函数 loss 对 $\boldsymbol{S}$ 的梯度，该梯度简记为 $\mathrm{d}\boldsymbol{S}$（注意之后的损失函数对参数的梯度都简记为 $\mathrm{d}\boldsymbol{W}$，省略重复的 dloss），注意 $\mathrm{d}\boldsymbol{S}$ 和 $\boldsymbol{S}$ 的尺寸相同。网络最后两层的模型如下：

$$\begin{gathered}\boldsymbol{A}_{n-1}=\boldsymbol{H}_{n-2}\boldsymbol{W}_{n-1},\ \boldsymbol{H}_{n-1}=f(\boldsymbol{A}_{n-1}),\\ \boldsymbol{S}=\boldsymbol{H}_{n-1}\boldsymbol{W}_{n},\\ \mathrm{loss}=\boldsymbol{L}(\boldsymbol{S})\end{gathered}\tag{6.11}$$

再次提醒此时所有变量都是矩阵，已经得到 $\mathrm{d}\boldsymbol{S}$。现在从 $\mathrm{d}\boldsymbol{S}$ 开始反向传播，推导其他变量的梯度。首先，根据 $\boldsymbol{S}=\boldsymbol{H}_{n-1}\boldsymbol{W}_n$ 可以得到 $\boldsymbol{H}_{n-1}$，$\boldsymbol{W}_n$ 的梯度，其值为：$\mathrm{d}\boldsymbol{H}_{n-1}=\mathrm{d}\boldsymbol{S}\boldsymbol{W}_n^{\mathrm{T}}$ 和 $\mathrm{d}\boldsymbol{W}_n=\boldsymbol{H}_{n-1}^{\mathrm{T}}\mathrm{d}\boldsymbol{S}$，其中矩阵上标 T 表示矩阵的转置。根据 $\boldsymbol{H}_{n-1}=f(\boldsymbol{A}_{n-1})$和 $\mathrm{d}\boldsymbol{H}_{n-1}$ 可得 $\mathrm{d}\boldsymbol{A}_{n-1}=\mathrm{d}\boldsymbol{H}_{n-1}f'(\boldsymbol{A}_{n-1})$，注意该式是逐元素计算的，不是矩阵乘法。现在有了 $\mathrm{d}\boldsymbol{A}_{n-1}$，再结合 $\boldsymbol{A}_{n-1}=\boldsymbol{H}_{n-2}\boldsymbol{W}_{n-1}$，可得 $\mathrm{d}\boldsymbol{H}_{n-2}=\mathrm{d}\boldsymbol{A}_{n-1}\boldsymbol{W}_{n-1}^{\mathrm{T}}$ 和 $\mathrm{d}\boldsymbol{W}_{n-2}=\boldsymbol{H}_{n-2}^{\mathrm{T}}\mathrm{d}\boldsymbol{A}_{n-1}$。有了 $\mathrm{d}\boldsymbol{H}_{n-2}$，可以继续往前推，反向得到各层参数和中间变量的梯度，直到得到第一层参数的梯度和输入的梯度。

这里的核心是矩阵乘法 $\boldsymbol{A}=\boldsymbol{H}\boldsymbol{W}$，并且已知 $\mathrm{d}\boldsymbol{A}$，求 $\mathrm{d}\boldsymbol{H}$ 和 $\mathrm{d}\boldsymbol{W}$，那么：

$$\begin{aligned}\mathrm{d}\boldsymbol{H} &= \mathrm{d}\boldsymbol{A}\boldsymbol{W}^{\mathrm{T}} \\ \mathrm{d}\boldsymbol{W} &= \boldsymbol{H}^{\mathrm{T}}\mathrm{d}\boldsymbol{A}\end{aligned} \tag{6.12}$$

希望读者记住此公式，该公式比较好记，首先把这些变量看成标量，则 $\mathrm{d}\boldsymbol{H}=\boldsymbol{W}\mathrm{d}\boldsymbol{A}$，$\mathrm{d}\boldsymbol{W}=\boldsymbol{H}\mathrm{d}\boldsymbol{A}$，然后把它们看作矩阵。由于矩阵乘法需要满足矩阵的尺寸一致性，即尺寸为 $m\times n$ 和尺寸为 $n\times k$ 的矩阵乘积为 $m\times k$ 的矩阵，两个相乘矩阵必须有个维度相等。可以发现，只有 $\mathrm{d}\boldsymbol{H}=\mathrm{d}\boldsymbol{A}\boldsymbol{W}^{\mathrm{T}}$ 和 $\mathrm{d}\boldsymbol{W}=\boldsymbol{H}^{\mathrm{T}}\mathrm{d}\boldsymbol{A}$ 的形式满足矩阵乘法的性质。

6.5　softmax 损失函数梯度计算

根据第 2 章的内容，可得 softmax 损失函数的定义：

$$\begin{aligned}p_k &= -\log(\mathrm{e}^{s_k} / \sum_{j=1}^{K}\mathrm{e}^{s_j}) \\ L_i &= -\log(p_{y_i})\end{aligned} \tag{6.13}$$

其中 y_i 是样本 i 的标签，即样本类别的序号，p_k 是第 k 类的概率，s_k 是第 k 类的分值。现在求 $\partial L_i / \partial s_k$，利用高数知识，可以很容易求出该梯度：

$$\partial L_i / \partial s_k = p_k - I(y_i = k) \tag{6.14}$$

其中，$I(y_i=k)$ 表示当 k 等于标签 y_i 时，则结果等于 1，否则为 0。这样我们就得到了 $\mathrm{d}\boldsymbol{S}$，然后就可以进行 6.4 节的梯度反向传播，获得各层参数的梯度。

式 (6.14) 十分简洁，同时具有启发意义。由于概率 p_k 大于 0，所以当 k 不等于标签 y_i 时，梯度为 p_k。梯度为正，表示分值 s_k 增加，损失值 L_i 随之增加，也即随着损失值 L_i 的减小，s_k 会减小。这正是我们所希望的：对于非类别对应的分值，希望其越小越好。当 k 等于标签 y_i 时，梯度是 $p_{y_i}-1$，小于 0。梯度为负，表示分值 s_{y_i} 增加，损失值 L_i 减小，也即损失值 L_i 减小，会使 s_k 增大。这正是我们所希望的：对于样本的类别对应的分值，我们希望其越大越好。

实践中，一般会对多个样本同时进行处理，代码如下：

```
    K = 10  # 类别数
    N = 128  # 样本数量
    scores = np.random.randn(N, K) # 一行一个样本的分值向量
    labels = np.random.randint(K, size = N)
    exp_scores = np.exp(scores)
    exp_scores_sum = np.sum(exp_scores, axis = 1, keepdims = True)
    probs = exp_scores/exp_scores_sum

    dscores = probs.copy()
①   dscores[range(N), labels] -= 1
    dscores /=  N
```

随机生成分值矩阵和样本标签，其中分值矩阵的一行是一个样本的分值向量。然后计算概率矩阵和分值矩阵的梯度，最后取平均。注意语句 ① 的索引方式，用于获得样本的类别分值。分值梯度 dscores 的尺寸和 scores 一致，每行对应一个样本。

6.6　全连接层梯度反向传播

全连接层梯度的计算和 6.4 节所述很像，只是多了偏置。公式为

$\boldsymbol{S}=\boldsymbol{X}\boldsymbol{W}+\boldsymbol{b}$，其中 $\boldsymbol{X}$ 是输入，$\boldsymbol{W}$ 是权重，$\boldsymbol{b}$ 是偏置，$\boldsymbol{S}$ 是输出分值矩阵。已知 d$\boldsymbol{S}$，需要计算 d$\boldsymbol{X}$、d$\boldsymbol{W}$ 和 d$\boldsymbol{b}$。由于 NumPy 的广播机制，应注意 $\boldsymbol{b}$ 是向量。代码如下：

```
    D = 784   # 数据维度
    K = 10   # 类别数
    N = 128   # 样本数量

    X = np.random.randn(N, D) # 一行一个样本
    W = 0.01 * np.random.randn(D,K)
    b = np.zeros((1,K))
    scores = np.dot(X, W) + b

    # 反向传播
    dscores = np.random.randn(N, K) # 与 scores 的尺寸一样
    dX = np.dot(dscores, W.T)
    dW = np.dot(X.T, dscores)
①   db = np.sum(dscores, axis = 0, keepdims = True)
```

首先前向计算分值矩阵，同样每一行是一个样本的分值向量。随机生成分值矩阵的梯度 dscores 后，计算 d$\boldsymbol{X}$ 和 d$\boldsymbol{W}$。需要特别注意语句 ① 中 d$\boldsymbol{b}$ 的计算方式，d$\boldsymbol{b}$ 中的每个元素是 dscores 每列之和。因为如果不使用 NumPy 的广播机制，则 $\boldsymbol{b}$ 向量需扩增为矩阵。计算 d$\boldsymbol{b}$ 的每个元素，有 N 条路径，N 条路径之和就是 d$\boldsymbol{b}$ 元素的值。

6.7　激活层梯度反向传播

当激活函数采用 ReLU 时，公式为 $\boldsymbol{H}_{\text{out}}=\text{ReLU}(\boldsymbol{H}_{\text{in}})$，其中 $\boldsymbol{H}_{\text{in}}$ 是激活层的输入，$\boldsymbol{H}_{\text{out}}$ 是输出。注意，该公式是逐元素计算的。已知 d$\boldsymbol{H}_{\text{out}}$，需要计算 d$\boldsymbol{H}_{\text{in}}$。ReLU 的梯度计算特别简单，因为当输入小于 0 时，梯

度为 0；当输入大于 0 时，梯度为 1。所以 $\boldsymbol{H}_{\text{in}}$ 的每个元素只需简单地和 0 比较，如果大于 0，则输出梯度等于输入梯度，否则为 0。代码如下：

```
dim1 = 164
dim2 = 128
Hin = np.random.randn(dim1, dim2)
Hout = np.maximum(0, Hin) # ReLU 激活
dHout = np.random.randn(dim1, dim2) # 与 Hout 的尺寸一样
# 反向传播
dHin = dHout
dHin[Hin <=  0] = 0
```

对于随机生成的输入矩阵 $\boldsymbol{H}_{\text{in}}$ 和梯度 $\text{d}\boldsymbol{H}_{\text{out}}$，先进行 ReLU 激活，输入梯度先赋值为输出梯度，然后进行梯度反传，即把输入值 $\boldsymbol{H}_{\text{in}}$ 小于等于 0 的梯度设置为 0。该代码十分清晰地展示了梯度反向传播，但是增加了很多中间变量，存储开销比较大。实践中代码可以简化如下：

```
    dim1 = 164
    dim2 = 128
    hidden_data = np.random.randn(dim1, dim2)
    hidden_data = np.maximum(0, hidden_data) # ReLU 激活
    dhidden_data = np.random.randn(dim1, dim2)
    # 与 hidden_data 的尺寸一样
    # 反向传播
①   dhidden_data[hidden_data <=  0] = 0
```

由于输入数据和输出数据的尺寸相同，又不需要存储中间结果，所以输入和输出数据可以共享存储空间，使用同一个变量。

与其他激活函数的梯度反向传播程序类似，只是语句 ① 有所不同，现在以 ELU 激活函数为例进行介绍，公式为：

$$f(x)=\begin{cases}x & \text{如果 } x\geqslant 0\\ \text{e}^{x}-1 & \text{如果 } x<0\end{cases}$$

其梯度为：

$$f'(x)=\begin{cases}1 & \text{如果}\, x \geqslant 0 \\ \mathrm{e}^x & \text{如果}\, x<0\end{cases}$$

而 $\mathrm{e}^x = f(x)+1$，所以语句①的代码为：`dhidden_data[hidden_data < =  0] * = (hidden_data + 1)`。

6.8　卷积层梯度反向传播

卷积层的梯度反向传播比较复杂，本书采用大矩阵乘法来实现。梯度反向传播的计算过程和卷积层的正向计算过程正好相反，是其逆过程，所以本节首先简单回顾一下如何利用大矩阵乘法实现卷积层的正向计算过程。其核心有三步。

(1) 将输入特征图变换为大矩阵 `matric_data`。

(2) 进行矩阵相乘和非线性激活（等价于全连接层）后得到 `filter_data`。

(3) 将 `filter_data` 变换为输出特征图。

卷积层梯度反向传播是已知输出特征图的梯度，求输入特征图的梯度及卷积核的梯度，其过程如下。

(1) 把输出特征图的梯度 `dout_data` 变换为矩阵形式（正向计算第 (3) 步的逆过程）。

(2) 将全连接层和激活层的梯度进行反向传播。

(3) 把第 (2) 步得到的大矩阵梯度变换为特征图形状的梯度，即得到输入特征图的梯度。

梯度反向传播时，需要正向计算过程得到的 matric_data 和 filter_data，读者只要理解了正向计算过程，其逆过程很简单，所以这里直接给出代码。若需要示意图，可以参考 4.2.5 节的图 4.4。

首先，定义一些超参数，同时随机生成一些必要的数据，这些数据能够在正向计算中获得：

```
filter_size = 3
filter_size2 = filter_size*filter_size
stride = 1
padding = (filter_size - 1)//2

batch = 8
(in_height, in_width, in_depth) = (32, 48, 16)
(out_height, out_width, out_depth) = (32, 48, 32)
out_size = out_height*out_width

dout_data = np.random.randn(batch, out_height, out_width,
    out_depth)
matric_data = np.random.randn(out_size*batch,
    filter_size2*in_depth)
filter_data = np.random.randn(out_size*batch, out_depth)
weights = 0.01 * np.random.randn(filter_size2*in_depth,
    out_depth)
```

其中，dout_data 是上次梯度反向传播得到的，是输入梯度，matric_data、filter_data 和 weights 都是在正向计算中得到的。

然后把 dout_data 变换为矩阵形式 dfilter_data，即尺寸与 filter_data 一致。dout_data 的每个深度列就是 dfilter_data 的一行，代码如下：

```
dfilter_data = np.zeros_like(filter_data)
for i_batch in range(batch):
    i_batch_size = i_batch*out_size
    for i_height in range(out_height):
        i_height_size = i_batch_size + i_height*out_width
        for i_width in range(out_width):
            dfilter_data[i_height_size + i_width, :] =
                dout_data[i_batch, i_height, i_width, :]
```

其次，进行激活层和全连接层反向传播，得到权重 dweights、偏置的梯度 dbias 以及矩阵形式的梯度 dmatric_data：

```
dfilter_data[filter_data <= 0] = 0

dweights = np.dot(matric_data.T, dfilter_data)
dbias = np.sum(dfilter_data, axis = 0, keepdims = True)
dmatric_data = np.dot(dfilter_data, weights.T)
```

最后，把 dmatric_data 变换为特征图形状的梯度，即得到输入特征图的梯度：

```
padding_height = in_height + 2*padding
padding_width = in_width + 2*padding
dpadding_data = np.zeros((batch, padding_height,
    padding_width, in_depth) )

height_ef = padding_height - filter_size + 1
width_ef = padding_width - filter_size + 1
for i_batch in range(batch):
    i_batch_size = i_batch*out_size
    for i_h, i_height in zip(range(out_height), range(0,
        height_ef, stride)):
```

```
            i_height_size = i_batch_size + i_h*out_width
            for i_w, i_width in zip(range(out_width), range(0,
               width_ef, stride)):
①              dpadding_data[i_batch, i_height : i_height +
                   filter_size,i_width : i_width + filter_size, :]
                       += dmatric_data[
                   i_height_size + i_w, :].reshape
                       (filter_size, filter_size, -1)

    if padding:
        din_data = dpadding_data[:,padding:-padding,padding:
            -padding,:]
    else:
        din_data = dpadding_data
```

其中，`dmatric_data` 的每一行数据就是 `dpadding_data` 的局部窗口的一个数据，注意行向量需要 reshape 成 3D 矩阵。特别注意语句①中的加号，这是因为 `dpadding_data` 数据有多条路径（即局部窗口有重叠）得到梯度，所以需要累加。

现在有了 `din_data`，又可以向前一层进行梯度反向传播了，这样一层一层传播下去，直到第一层。

6.9 最大值池化层梯度反向传播

最大值池化层梯度反向传播类似于 ReLU 激活层梯度反向传播。ReLU 是求 $\max(0, x)$的偏导数，最大值池化层是求 $f = \max(a, b, c, d)$的偏导数，其中 a、b、c 和 d 是输入 2×2 窗口的数据。该函数的偏导数很简单，对 a、b、c 和 d 中最大值的梯度为 1，其他为 0。计算过程如下。

根据定义，对变量 a 的偏导数为：

$$[\max(a+\mathrm{d}a, b, c, d) - \max(a, b, c, d)]/\mathrm{d}a$$

当 a 是最大值时，上式变为 $(a+\mathrm{d}a-a)/\mathrm{d}a=1$；当 a 不是最大值时，假设此时 b 是最大值，则上式变为：$(b-b)/\mathrm{d}a=0$。所以进行最大值池化层梯度反向传播时，只需记录输入特征图的每个局部窗口的最大值即可。如果该元素是最大值，则该位置处的输出梯度等于输入梯度，否则为 0。

首先，定义一些超参数并随机生成一些必要的数据，这些数据能在正向计算中获得。其中，dout_data 是上次梯度反向传播得到的，是输入梯度。matric_data_max_pos 记录了输入特征图中最大值的位置，尺寸是变换后的大矩阵，每行只有局部窗口 4 个元素，它是从正向计算中得到的。相关代码如下：

```
   (batch, in_height, in_width, in_depth) = (8, 32, 48, 16)
   filter_size = 2
   filter_size2 = filter_size*filter_size
   stride = 2
   out_height = (in_height - filter_size)//stride + 1
   out_width = (in_width - filter_size)//stride + 1
   out_depth = in_depth
   out_size = out_height*out_width

   dout_data = np.random.randn(batch, out_height, out_width,
       out_depth)

①  matric_data_max_pos = np.random.randn(out_size*in_depth*
       batch, filter_size2)
②  matric_data_max_pos = matric_data_max_pos > 0
③  matric_data_not_max_pos = ~matric_data_max_pos
```

```
din_data = np.zeros((batch, in_height, in_width, in_depth),
    dtype = np.float64)
```

其中，语句 ① 生成随机数据，语句 ② 模拟最大值位置，语句 ③ 对 matric_data_max_pos 取反，即得到非最大值的位置，因为非最大值位置处的梯度为 0。din_data 是输入特征图的梯度，是需要计算的。

然后遍历每一个特征图的局部窗口，将 din_data 最大值位置处的梯度赋值为 dout_data，将非最大值位置处的梯度赋值为 0：

```
    height_ef = in_height - filter_size + 1
    width_ef = in_width - filter_size + 1
    for i_batch in range(batch):
        i_batch_size = i_batch*out_size*in_depth
        for i_h_out, i_height in zip(range(out_height), range(0,
            height_ef, stride)):
            i_height_size = i_batch_size +
                i_h_out*out_width*in_depth
            for i_w_dout, i_w, i_width in zip(range(out_width),
                                            range(0, in_depth*out_
                                            width, in_depth),
                                            range(0, width_ef,
                                            stride)):
④               md = matric_data_not_max_pos[i_height_size +
                    i_w : i_height_size + i_w + in_depth, : ]
⑤               din = din_data[i_batch, i_height : i_height
                    + filter_size, i_width  : i_width  +
                    filter_size, :]
⑥               dout = dout_data[i_batch, i_h_out, i_w_dout, :]
                for i in range(filter_size):
                    for j in range(filter_size):
⑦                       din[i, j, :] = dout[:]
⑧                       din[i, j, :][md[:, i*filter_size + j]] = 0
```

在上述代码中，前面的语句用于遍历每个窗口，语句 ④ 获得局部窗口中（注意是包含整个深度维度）非最大值位置的 md，它是二维矩

阵；语句 ⑤ 获得输入梯度局部窗口中的数据（也是整个深度维度）`din`，它是三维矩阵；语句 ⑥ 获得输出梯度局部窗口数据（整个深度维度）`dout`，它是一维矩阵。然后对局部窗口中每个深度列进行赋值。语句 ⑦ 是 `din` 深度列复制 `dout` 的值，语句 ⑧ 对非最大值位置的梯度置 0，需要特别注意其索引方式。

上面介绍了常规神经网络和卷积网络中各种层的梯度反向传播算法及代码，希望读者认真掌握。根据链式法则和梯度反向传播原则，如果以后碰到新的网络结构，大家应该能实现梯度反向传播，比如 Inception 和 ResNet 结构。

第三部分

实战篇

第 7 章

训练前的准备

通过前面两部分的学习，我们已经掌握了机器学习最常用的线性模型、神经网络模型和卷积网络模型，以及基于梯度下降法的各种优化方法和梯度反向传播算法。有了这三方面的知识，理论上就可以开始训练能解决实际问题的模型了。但在开始训练前，还有一些细节问题需要解决，这些问题看起来不起眼，但对最终的训练结果有比较重要的影响，希望读者重视。

7.1 中心化和规范化

在现实任务中，样本属性各种各样，大小各不相同，如果直接使用原始数值进行机器学习，效果可能会比较差，甚至不能进行有效学习，所以必须先对数据进行预处理。为了使读者理解为什么需要数据预处理，以及需要什么样的预处理，本节将采用二分类的线性模型进行理论解释。线性模型是神经网络和卷积网络模型的基本构件，所以这些理论解释同样适用于神经网络和卷积网络模型。

7.1.1　利用线性模型推导中心化

假设样本有 3 个属性，则二分类的线性模型为：$y = w_1x_1+w_2x_2+w_3x_3+b$，其中 $w_i(i = 1, 2, 3)$ 是权重，b 是偏置，y 是模型输出。因为要得出最优化模型，所以参数只有在实数域取值，才能最好地与样本数据一致。如果参数只在有理数或整数域取值，显然没有实数域好，所以权重和偏置参数都是实数，这样输出 y 也是实数。在进行样本预测时，需要根据 y 进行二分类，最合理的做法是将 y 与某个固定值比较，大于该固定值，则判断样本为正类，否则为负类。由于 y 的取值在实数域，所以该固定值最合理的取值是 0。这样所有样本都与 0 进行比较，正样本输出大于 0，负样本输出小于 0，当正负样本数量相等时（实际情况下，大部分是这样），则输出 y 的期望就是 0。输出 y 的绝对值越大，表示对样本分类正确的信心就越大。

如果输出 y 期望为 0，那么会对变量 $x_i(i = 1, 2, 3)$ 提出要求。因为参数 w_i 是固定值，显然每个变量 x_i 都应该是 0 均值，这样输出 y 的期望才能为 0（假定偏置 b 为 0）。使变量 0 均值化的预处理过程称为中心化，它是数据预处理最常用的操作。中心化是每个样本的特征属性减去所有样本（正负样本）的对应特征属性的均值，是逐属性操作。代码如下：

```
D = 784  # 数据维度
N = 128  # 样本数量
X = np.random.randn(N, D) # 一行一个样本
X -= np.mean(X, axis = 0)
```

其中，`X` 是所有样本数据的矩阵，每行一个样本；`np.mean(X, axis = 0)`用于计算所有样本中每个属性的均值。

7.1.2 利用属性同等重要性推导规范化

当采用迭代法进行模型优化时（梯度下降法就是一种迭代法），需要对参数进行初始化。假设我们对学习任务一无所知，即没有任何先验知识，不知道变量 x_i 对分类是正相关还是负相关，因此对参数最合理的初始化是 0。为了打破对称性，权重初始化为均值 0 的小随机数。特别地，偏置 b 基本都直接初始化 0。

当采用梯度下降法进行优化时，每个参数的更新方式是一致的，所以我们希望参数的最优值范围大致相当，这样能使所有参数同步收敛，提高学习效率。而由于 w_i 的最优值范围大致相当，变量 x_i 通过与参数相乘影响输出，所以假设我们没有任何先验知识，只能假定每个变量 x_i 对分类的影响程度一致，那么必然要求每个变量 x_i 的取值范围一致。如果某个变量的取值范围明显偏大，会产生前后矛盾的问题。例如，当 x_1 的取值范围是 x_2 的 10 倍时，各 w_i 的取值范围一致。假设 w_1 等于 w_2，这样 w_1x_1 的值是 w_2x_2 的 10 倍，x_1 对输出 y 的影响也是 x_2 的 10 倍，即 x_1 是更重要的影响因素。显然这与“每个变量 x_i 对分类的影响程度一致”的假设矛盾，所以我们必然要求每个变量 x_i 的取值范围一致，这就是规范化操作。最常用的规范化是除以标准差，这个操作与中心化操作一样，是逐属性的，标准差是通过计算所有样本得到的。注意规范化之前一定要进行中心化。代码如下：

```
D = 784
N = 128
X = np.random.randn(N, D)
X -= np.mean(X, axis = 0)
X /= np.std(X, axis = 0)
```

中心化和规范化的组合操作就是对随机变量的标准化，变为均值为 0，方差为 1 的分布。数学公式为：

$$\hat{x} = (x - \mu) / \sigma \tag{7.1}$$

其中 μ 为均值，σ 是标准差。

中心化和规范化组合操作的另一种常见方式是把每个属性的取值都统一到 $[-1, 1]$ 这个区间内。这个方法利用每个属性的最大值和最小值可以很容易实现，代码如下：

```
D = 784
N = 128

X = np.random.randn(N, D)
minX = np.min(X, axis = 0)
maxX = np.max(X, axis = 0)

X = (X - minX)/(maxX - minX)
X = 2*X - 1
```

规范到 $[-1, 1]$ 的方法比较容易受到噪声或异常值的影响，因为噪声或异常值会影响属性的最大值或最小值。

规范化的假设认为每个属性的重要程度一致，如果有先验知识能表明某些属性更重要或更不重要，则其规范化的范围就要随之做出调整，增大或减小其影响。具体变换到多大范围是由算法的超参数决定，需要利用大量的计算资源不断地试错。所以在实践中，如果没有明确哪个属性特别重要或不重要，还是应规范化到同一范围，因为之后还可以通过调整参数 w_i 的取值来控制变量的影响程度，只是此时参数的收敛速度可能不同步。

7.1.3 中心化和规范化的几何意义

中心化的几何意义是平移样本点云，使样本点云的重心与原点重合，这样分类超平面就在原点附近，比较容易学习。如果不进行中心化，样本点云的重心可能远离原点，分类超平面也可能远离原点，导致偏置 b 的绝对值可能远大于 0。因为 b 初始化为 0，而最优 b 远离 0，所以在使用基于梯度下降法等局部方法时，要学习出如此大的差值是很困难的。

规范化的几何意义是对点云进行拉伸，使数据点云完全处于高维正方形内。它的物理意义是去除每个属性量纲的影响，例如属性是质量，则取不同单位时（如千克和毫克），数值会差别很大，但分类的效果不应受单位的影响，故必须去除量纲的影响，进行规范化。

7.2 PCA 和白化

PCA 和白化也是一种常用的数据预处理方法。实际任务中，变量之间可能会存在复杂的依赖关系，这种关系可能是非线性或线性的。例如 x_1 增大，x_2 随之增大，这表明这两个变量存在正相关。这必然会对权重 w_1 和 w_2 产生约束，但优化算法难以处理这种“隐形”约束。所以希望在优化之前，去除变量之间的相关性，使学习变得更容易。PCA 就是一种去除变量之间的相关性，使之线性无关的最常用方法。特别强调的是，PCA 只能去除线性相关性，不能去除非线性相关性。线性相关性是指变量之间满足线性函数关系，非线性相关性是指变量之间满足函数关系，但不是线性函数。

7.2.1　从去除线性相关性推导 PCA

先从最简单的情况入手，假设样本只有两个属性 x 和 y，我们采集了一系列样本 (x_i, y_i)，把它们表示为数据矩阵 $\boldsymbol{D}$，其中每行为一个样本数据，每列为一个属性取值，设两列向量为 $\boldsymbol{X}$ 和 $\boldsymbol{Y}$。如果这两个属性之间存在线性关系，则它们可以表示为 $y=kx+b$，其中系数 $k\neq 0$（如果 $k=0$，那么这两个属性线性无关）。可以利用最小二乘法进行线性拟合，得到系数：

$$k=1/m\sum_{i=1}^{m}(x_i-\overline{x})(y_i-\overline{y})/[m\sum_{i=1}^{m}x_i^2-(\sum_{i=1}^{m}x_i)^2] \tag{7.2}$$

令 $\mathrm{cov}(x,y)=1/m\sum_{i=1}^{m}(x_i-\overline{x})(y_i-\overline{y})$，显然，当 $\mathrm{cov}(x, y)=0$ 时，x 和 y 线性无关；当 $\mathrm{cov}(x,y)>0$ 时，x 和 y 线性正相关；当 $\mathrm{cov}(x,y)<0$ 时，x 和 y 线性负相关。因为 $\mathrm{var}(x)=\mathrm{cov}(x, x)$ 是变量 x 的方差，所以称 $\mathrm{cov}(x,y)$ 为协方差。

现在解决了两个变量线性无关的度量，即协方差为 0。那么，是否存在一个线性变换 $\boldsymbol{C}$，对属性进行变换，使新变量的协方差为 0 呢？为了简化书写，下面假设已经对属性进行了中心化，即每个属性的均值为 0，所以协方差 $\mathrm{cov}(x,y)=1/m\sum_{i=1}^{m}x_iy_i$，这个刚好是内积，$\mathrm{cov}(x,y)=1/m\,\boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}$。线性变换 $\boldsymbol{C}$ 对数据矩阵 $\boldsymbol{D}$ 进行变换，变换后的新数据矩阵为 $\boldsymbol{D}'=\boldsymbol{DC}=[\boldsymbol{DC}_1,\boldsymbol{DC}_2]$，新的列向量为 $\boldsymbol{X}'=\boldsymbol{DC}_1$，$\boldsymbol{Y}'=\boldsymbol{DC}_2$，其中 $\boldsymbol{C}_1$ 和 $\boldsymbol{C}_2$ 是 $\boldsymbol{C}$ 的列向量，则新向量的方差为：$\boldsymbol{X}'^{\mathrm{T}}\boldsymbol{X}'=\boldsymbol{C}_1^{\mathrm{T}}\boldsymbol{D}^{\mathrm{T}}\boldsymbol{DC}_1$，$\boldsymbol{Y}'^{\mathrm{T}}\boldsymbol{Y}'=\boldsymbol{C}_2^{\mathrm{T}}\boldsymbol{D}^{\mathrm{T}}\boldsymbol{DC}_2$，协方差为 $\boldsymbol{X}'^{\mathrm{T}}\boldsymbol{Y}'=\boldsymbol{C}_1^{\mathrm{T}}\boldsymbol{D}^{\mathrm{T}}\boldsymbol{DC}_2$。矩阵 $\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}$ 是维度为

2×2 的方阵，维数等于属性数量。该矩阵的对角线元素为原始属性的方差，非对角线元素为协方差，故该矩阵被称为协方差矩阵，因为 $(\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D})^{\mathrm{T}}=\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}$ 是对称矩阵。令 $\boldsymbol{C}_1^{\mathrm{T}}\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{C}_1=\lambda_1$，$\boldsymbol{C}_2^{\mathrm{T}}\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{C}_2=\lambda_2$，这两个数分别是两个新属性的方差，它们大于等于 0。因为新属性需线性无关，所以协方差 $\boldsymbol{X}'^{\mathrm{T}}\boldsymbol{Y}'=\boldsymbol{C}_1^{\mathrm{T}}\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{C}_2=0$。仔细分析这 3 个等式，可以发现向量 $\boldsymbol{C}_1$ 和 $\boldsymbol{C}_2$ 有正交的感觉，再结合矩阵的特征向量正交的性质，就可以求出变换矩阵 $\boldsymbol{C}$ 了。如果把向量 $\boldsymbol{C}_1$、$\boldsymbol{C}_2$ 与对应的 λ_1 和 λ_2 看作协方差矩阵的特征向量和特征值，即 $\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{C}_1=\lambda_1\boldsymbol{C}_1$，$\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{C}_2=\lambda_2\boldsymbol{C}_2$，代入上面 3 个等式得 $\boldsymbol{C}_1^{\mathrm{T}}\boldsymbol{C}_1=1$，$\boldsymbol{C}_2^{\mathrm{T}}\boldsymbol{C}_2=1$，$\boldsymbol{C}_1^{\mathrm{T}}\boldsymbol{C}_2=0$，即要求线性变换 $\boldsymbol{C}$ 为正交矩阵，所以只需把特征向量正交化就可以了。上面分析过程的样本只有两个属性，如果有 d 个属性，其推理过程完全一样，故不重复。

现在总结一下，如何去除样本属性之间的线性相关性。假设样本有 d 个属性，我们采集了 n 个样本，数据矩阵为 $\boldsymbol{D}$，其中每行是一个样本，则 $\boldsymbol{D}$ 的尺寸为：$n\times d$。第一步计算协方差矩阵 $\mathrm{COV}=\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}$，对角线元素为属性的方差，非对角线元素为协方差。第二步对协方差矩阵 COV 进行特征值分解，得到特征值和对应的特征向量$(\lambda_i,\boldsymbol{C}_i)$，其中特征向量需要规范化，$\boldsymbol{C}_i^{\mathrm{T}}\boldsymbol{C}_i=1$，$\boldsymbol{C}_i^{\mathrm{T}}\boldsymbol{C}_j=0$，即向量长度为 1 且两两正交。第三步利用向量 $\boldsymbol{C}_i$ 得到新属性 $\boldsymbol{D}\boldsymbol{C}_i$，新属性的方差为 λ_i（大于等于 0），协方差为 0，此时数据矩阵 $\boldsymbol{D}'=\boldsymbol{D}\boldsymbol{C}$。由于矩阵 $\boldsymbol{C}$ 是正交矩阵，所以变换是可逆的，可得 $\boldsymbol{D}=\boldsymbol{D}'\boldsymbol{C}^{\mathrm{T}}$。

7.2.2 PCA 代码

实践中，一般用奇异值分解代替特征值分解，以获得更稳定的数

值解和更快的速度。奇异值分解的推导和理解均涉及复杂的矩阵知识，本节直接给出代码，不进行解释：

```
    D = 784
    N = 128
    X = np.random.randn(N, D)
    mean = np.mean(X, axis = 0)
    X -= mean
    COV = np.dot(X.T, X)/X.shape[0] # 协方差矩阵
①   C,S,V = np.linalg.svd(COV)
②   X_decor = np.dot(X, C)
```

其中，X_decor 就是去除线性相关之后的数据矩阵，新属性之间不存在线性相关，即 X_decor 的协方差矩阵为对角矩阵。X_decor 的尺寸和 X 一样，每行是一个样本，每列是所有样本的新属性值。X_decor 是中心化的，即每个属性的均值为 0。语句 ① 进行奇异值分解，得到正交矩阵 $\boldsymbol{C}=(\boldsymbol{C}_1, \boldsymbol{C}_2, \cdots, \boldsymbol{C}_d)$ 和特征值向量 $\boldsymbol{S}=(\lambda_1, \lambda_2, \cdots, \lambda_d)$。其中，$\boldsymbol{S}$ 中的特征值是降序排列的，所以 X_decor 中第一个属性的方差最大，后面依次减小。

7.2.3　PCA 降维

如果协方差矩阵的某个特征值为 0，说明该属性的方差为 0，属性值是恒定值，即各个样本在该属性上没有任何差别，那么该属性对分类没有任何价值，可以去掉该属性，这就是 PCA 降维原理。实际中，由于存在数值计算误差，方差很小的新属性都可以去掉，留下方差较大的新属性，实现数据降维。由于均值为 0、方差很小的属性的数值都十分接近 0，故可近似为 0，因此可以去除该属性。当方差为 0 时，即 $\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}\boldsymbol{C}_i=0$，其中特征向量 $\boldsymbol{C}_i$ 不为 0，故矩阵 $\boldsymbol{D}^{\mathrm{T}}\boldsymbol{D}$ 的秩小于 d，各个

列向量存在线性相关。

对于上一小节的程序，假设希望留下 127 个属性，去掉剩下的 784 − 127 = 657 个属性，只需加上代码 `X_decor_reduce = X_decor[:, :127]`，即留下方差最大的 127 个属性。实际中，不需要先计算 `X_decor` 再降维，可以直接得到降维后的 `X_decor_reduce`，即只需把语句 ② 改为 `X_decor_reduce = np.dot(X, C[:, :127])`。

数据降维也实现了数据压缩，原来样本需要 d 个属性描述，现在只需更少的属性描述，存储量减小了。由于变换是可逆的，因此降维后的数据能恢复到原始数据。代码如下：

```
①   X_decor_reduce = np.dot(X, C[:, :127])
②   zero_matic = np.zeros( (X.shape[0], X.shape[1]-127) )
③   X_decor_reduce_zero = np.hstack( (X_decor_reduce,
        zero_matic) )
④   X_denoise = np.dot(X_decor_reduce_zero, C.T)
⑤   X_denoise += mean
```

其中，语句 ① 进行降维，语句 ② 和语句 ③ 去除近似 0 的属性数值，语句 ④ 进行逆变换，语句 ⑤ 加上原均值，得到降维后重构的数据矩阵。

对于一个实际任务，留下的属性个数是算法超参数。属性取得少，可以减小样本数据大小，节约存储空间，同时提高训练和预测速度。但缺点是会损失样本中的一些有用信息，这部分信息虽少，但对学习效果也是有影响的。注意，方差小的属性也可能含有样本差异的重要信息，因为计算方差是采用原始属性数值，会受所用单位影响，比如属性采用厘米作为单位，方差为 1，如果换成米为单位，则方差变为 `10**(-4)`，缩小了很多倍。因此，在实践中，只有样本属性特别多，

或者属性之间存在明显的线性相关时，才采用 PCA 降维。

具体降维时，一般采用经验法则。一是观察法，即画出方差 S 的曲线，当曲线十分接近 x 轴时，取此点之前的属性。如果运行上面的程序，会发现第 128 个方差十分小，所以只需留下前 127 个属性。请读者思考为什么是前 127 个方差大。

另一种方法是比例法，即降维后的数据信息量占原始信息量的比例要大于阈值（如 95%），如何衡量数据信息量呢？这时可以利用方差衡量，属性方差越大表示该属性的信息量越大。具体做法是：计算前 d' 个方差之和占总方差之和的比例，取最小的 d' 使比例大于 95%，则保留前 d'个属性即可。

换一个角度考虑，属性的方差很小，一般是受噪声影响，否则理论上应该是 0，所以去除方差小的属性，是一种数据去噪技术。

7.2.4　PCA 的几何意义

PCA 的几何意义是：先对数据点云进行平移（减去均值），使重心和原点重合，然后旋转点云（乘以正交矩阵），使 `x_decor`“内嵌”到尽可能低的维度空间内。比如：三维空间中两个数据点组成的点云，因为两点能构成一个平面，所以首先平移这两点，使其中点与原点重合，然后旋转这两点构成的任一平面，使平面处于水平面内，这样这两点就处于二维空间内，而不是三维空间。再进一步，可以在水平面内旋转这两点，使之与 x 轴重合，内嵌到一维空间。同理，只要三维空间的点云位于一个平面内，总是能通过平移和旋转操作，使之位于

水平面内。点云内嵌到低维空间会带来一个好处：可以用更少的坐标来表示点。如上面的例子，原始的点位于三维空间，则需要 3 个分量 (x, y, z) 表示；内嵌到二维空间，就只需用 (x, y) 表示；最后到一维空间，仅需 x 表示了。这就是 PCA 的降维。

7.2.5 白化

注意，PCA 只是对数据点云进行了平移和旋转，数据点云的形状并没有改变。如果需要对属性进行规范化（去除单位量纲的影响），则点云需要改变形状，这时可以利用属性标准差进行规范化，即属性数值除以对应的标准差即可，其代码如下：

```
X_white = X_decor/np.sqrt(S + 10**(-5))
```

这种规范化也称为白化，几何意义是使每个属性的方差相同，数据点云完全处于高维球内。`X_white` 的协方差矩阵为单位矩阵，`X_white` 变成近似独立同分布（i.i.d）的数据，即每个属性都是均值为 0、方差为 1 的分布（近似同分布），任何属性之间的协方差为 0，没有线性相关性（近似独立）。独立同分布的数据十分有利于机器学习，可以使优化算法更快找到更好的参数解。特别强调下，规范化后的数据是近似同分布但没有去除线性相关性（无旋转操作）。

分母中加小常数 `10**(-5)`，是为了防止被除数为 0。因为当协方差矩阵不满秩时，存在为 0 的方差。白化最大的缺点是会扩大噪声，因为它把所有属性的标准差都拉伸为 1，包括方差很小的属性（很可能是噪声引起的）。小常数能达到去噪的目的，因为常数越大去噪能力越强，但数据也会被过度平滑。所以，最优小常数需要交叉验证。

7.3　卷积网络在进行图像分类时如何预处理

数据预处理是十分重要的步骤，它能加快学习算法的收敛速度，并且收敛到更优的点。

对于采用卷积网络进行图像分类的任务来说，由于图像像素已经规范化到 $[0, 255]$，输出的分值向量可以不是 0 均值的，所以不会使用 PCA 和白化。对图像的预处理很简单：或者规范化到 $[0, 1]$，只需每个像素除以 255；或者规范化到 $[-1, 1]$，只需每个像素先减去 128，再除以 128。当训练集图像足够多、分布足够广时，每个像素的均值会趋近 128，所以在实际中心化时，并不需要求像素的均值，可以用 128 代替实际均值。目前在实践中，采用 IMAGENET 数据库时，精确的值是 `MEAN_BGR = [103.062623801,115.902882574,123.151630838]`，它们都比较接近 128，其中红色分量均值最接近 128，蓝色最小。

样本一般划分为训练集、验证集和测试集 3 个子集，数据预处理中需要的一些统计数据（比如均值、方差和协方差矩阵）只能采用训练集进行计算，然后利用这些统计数据对验证集和测试集进行预处理。不能先利用所有样本进行预处理后，再划分为 3 个子集。因为在训练模型时，只能看到训练集，即只能用训练集的数据提取统计信息，验证集和测试集在训练模型的时候是不可见的，所以不可能利用这些训练时“不存在”的样本提取统计信息，然后再利用这些信息进行学习。以上是实践中比较容易犯的错误，需要读者注意。

7.4 BN

数据预处理是指对神经网络的输入层数据进行逐属性规范化的过程，数据经过网络的多层变换后（其中包括非线性变换），不再是规范化的，即均值不为 0，方差不是 1。比如，经过 ReLU 变换后，输出属性都大于或等于 0，均值肯定会大于 0，这样对于网络中的隐含层来说，学习就变困难了。那么，能不能对任意隐含层的输入数据进行规范化，提高收敛速度和学习效果呢？BN（Batch Normalization，批量规范化）就是一种著名的对隐含层输入数据进行规范化的方法。

7.4.1 BN 前向计算

BN 的操作方式和数据预处理中的中心化和规范化操作一样，都是逐属性进行的，先减去均值，再除以标准差。公式为：$\hat{x}=(x-\mu)/\sigma$。这里的均值和标准差是怎么计算的呢？数据预处理时的均值和标准差是整个训练集的均值和标准差，但是对于 BN 而言，不可能得到整个样本集的均值和标准差，只能退而求其次，采用批量的均值和标准差（即在使用批量梯度下降法时，每次迭代的样本子集），详见 5.1 节。经过 BN 后，每个隐含层的输出数据都规范成均值为 0、方差为 1 的分布，但同时会带来一个缺点：网络的表达能力受到限制。因为不管网络前面的层对输入数据进行怎样的变换，最后都变为均值为 0、方差为 1 的分布，这显然不合理，我们希望不同的层能学到不同的均值和方差，以适应网络的变化。因此，BN 引入两个可学习的参数 γ 和 β 来学习标准差和均值。最终，BN 公式为：

$$\hat{x} = (x - \mu) / \sigma$$
$$y = \gamma \hat{x} + \beta \tag{7.3}$$

γ 初始化为 1，β 初始化为 0。这样 BN 层输出数据的均值为 β，标准差为 γ，与网络前面的层无关，减小了网络中各层的耦合，有利于学习。而且 γ 和 β 是可学习的，增加了网络的自适应性。因此，增加 BN 层的网络能极大地提高收敛速度。代码如下：

```
    D = 784   # 特征维度
    batch = 32   # 批量数
    BN_EPSILON = 10**(-8)
①  beta = np.zeros(D)
②  gamma = np.ones(D)

    X_batch = 0.1*np.random.randn(batch, D)

    mu = np.mean(X_batch, axis = 0)
    var = np.var(X_batch, axis = 0)

    X_batch -= mu
    X_batch /= np.sqrt(var + BN_EPSILON)

③  Y = gamma*X_batch + beta
```

语句 ① 和 ② 用于初始化参数，这里 beta 初始化为 0，gamma 初始化为 1，它们是逐属性的。输入数据矩阵 X_batch 是批量数据，每行一个样本，每列一个属性数值。计算每个属性的均值 mu 和方差 var，进行标准化后，语句 ③ 进行再尺度化，得到输出数据 Y。注意 Y 是逐属性的，加小常数 BN_EPSILON 是防止除数为 0。

7.4.2　BN 层的位置

BN 层通常位于非线性激活层之前、全连接层之后，注意必须在激活前对数据进行规范化。加入 BN 层的典型网络结构如下：

$$
\begin{aligned}
&\boldsymbol{X}=\boldsymbol{U}\boldsymbol{W}\\
&\boldsymbol{Y}=\boldsymbol{BN}(\boldsymbol{X};\ \gamma,\beta)\\
&\boldsymbol{Z}=f(\boldsymbol{Y})
\end{aligned}
\tag{7.4}
$$

其中 $\boldsymbol{U}$ 是输入数据。因为 BN 层中的 β 起到偏置的作用，所以全连接层中不再需要偏置参数。$\boldsymbol{X}$ 是全连接层的输出，也是 BN 层的输入，$\boldsymbol{Y}$ 是 BN 层的输出，最后进行非线性激活，得到输出 $\boldsymbol{Z}$。

7.4.3 BN 层的理论解释

BN 层最显著的效果是收敛速度得到了极大提高，比如 14 倍的提速，同时能提高学习效果，也具有一定的正则化作用。所以增加了 BN 层后，可以设置更大的学习率，减小 L2 正则化系数，加快学习率衰减速度。

为什么 BN 层有这些效果呢？读者要充分理解其中道理，需深刻掌握 2.2 节和 6.3 节的内容。简而言之，第一，如果网络的权重都增加 $\lambda(\lambda>1)$ 倍，由于损失对权重尺度不具有不变性，损失会变小，这会增加网络学习的难度。第二，损失对权重的梯度与权重大小成正比，这样大权重的更新量大，容易导致其振荡，不容易收敛。BN 层刚好可以克服这两点。首先证明 BN 层具有权重尺度不变性：$\mathrm{BN}(\boldsymbol{XW})=\mathrm{BN}(\boldsymbol{X}(\lambda\boldsymbol{W}))$，即权重改变 λ 倍，不会影响其输出，这样最终的损失值也不会受到影响。证明很简单，因为在权重改变时，对应的均值和标准差均等比例改变：

$$
\begin{aligned}
\mathrm{BN}(\boldsymbol{X}(\lambda\boldsymbol{W}))&=\gamma\frac{(\boldsymbol{X}(\lambda\boldsymbol{W})-\mu')}{\sigma'}+\beta\\
&=\gamma\frac{(\lambda\boldsymbol{XW}-\lambda\mu)}{\lambda\sigma}+\beta=\mathrm{BN}(\boldsymbol{XW})
\end{aligned}
\tag{7.5}
$$

其次，由于

$$(\lambda \boldsymbol{W})\frac{\partial \mathrm{BN}(\boldsymbol{X}(\lambda \boldsymbol{W}))}{\partial(\lambda \boldsymbol{W})}=(\lambda \boldsymbol{W})\frac{1}{\lambda}\frac{\partial \mathrm{BN}(\boldsymbol{X}\boldsymbol{W})}{\partial \boldsymbol{W}}=\boldsymbol{W}\frac{\partial \mathrm{BN}(\boldsymbol{X}\boldsymbol{W})}{\partial \boldsymbol{W}} \tag{7.6}$$

权重更新量与权重的尺度 λ 无关，不会引起大权重的振荡。

由于 BN 层具有上面两个性质，所以对权重的初始化不是特别敏感，要求就不是那么高了。

7.4.4　BN 层在实践中的注意事项

BN 层在计算时需要用到批量数据的均值和方差等统计信息，理论上它们是全体样本统计信息的无偏估计。但由于每次批量的不同导致这些信息波动较大，所以实践中批量大小不能太小，一般要大于 32。

模型测试时，一般一次只测试一个样本，这时批量为 1，此时 BN 层的均值和方差如何计算呢？解决办法很简单，对训练过程中所有批量的均值和方差取移动平均，采用这些移动平均值作为模型测试时的均值和方差，公式如下：

```
variable = variable * decay + value * (1 - decay)
```

其中，`value` 是当前批量的统计信息（均值或方差），`variable` 是移动平均（如果是 `mean`，则初始化为 0，`var` 则初始化为 1），`decay` 是衰减指数，取值十分接近 1，常采用 0.9、0.99 或 0.999。

7.4.5 BN 层的梯度反向传播

BN 层是可微函数，所以可以采用梯度下降法进行优化，采用链式法则计算梯度。为了方便读者查阅，下面再次把前向代码列出：

```
D = 784
batch = 32
decay = 0.997
BN_EPSILON = 10**(-8)
beta = np.zeros(D)
gamma = np.ones(D)
X_batch = np.random.randn(batch, D)

① mu = np.mean(X_batch, axis = 0)
② var = np.var(X_batch, axis = 0)
③ std = np.sqrt(var + BN_EPSILON)
④ X_batch_hat = (X_batch - mu)/std
⑤ Y = gamma*X_batch_hat + beta
```

前向计算是已知输入 X_batch，求输出 Y；反向传播是已知 dY，求 dX_batch、dgamma 和 dbeta，这是一个十分好的练习链式法则的例子，请读者仔细研读：

```
dY = np.random.randn(batch, D) # 和 Y 的形状一致
# 反向传播语句⑤
dX_batch_hat = gamma*dY
dbeta = np.sum(dY, axis = 0)
dgamma = np.sum(X_batch_hat*dY, axis = 0)

# 反向传播语句④
dX_batch = dX_batch_hat/std
dmu = -np.sum(dX_batch_hat, axis = 0)/std
dstd = -1/(std*std) * np.sum((X_batch - mu)*dX_batch_hat,
    axis = 0)

# 反向传播语句③
```

```
dvar = 1/2*(var**(-0.5))*dstd

# 反向传播语句②
dX_batch += 2*(X_batch - mu)/batch * dvar

# 反向传播语句①
dX_batch += dmu/batch
```

在进行反向传播时，一定要注意广播机制和 `dX_batch` 的加号。

7.4.6　BN 层的地位探讨

有的文章把 BN 层作为网络的基础构件，和全连接层、非线性激活层的地位一样，我认为这明显抬高了 BN 层的地位。BN 层是对输入数据进行线性变换，前向代码的语句 ④ 和语句 ⑤ 清晰地展示了这点，所以它完全可以并入全连接层，并不能作为一个基础的组成部分。把 BN 层看作一种辅助优化算法的手段更合适，特别是针对梯度下降法，是一种自适应的重新参数化方法。在深度学习三巨头之一 Yoshua Bengio 的巨著《深度学习》中，就把 BN 层作为一种优化技巧。并且，目前也发展了很多类似 BN 的优化算法，如横向规范化（Layer Normalization）、权重规范化（Weight Normalization）、余弦规范化（Cosine Normalization）等，它们都是一种优化技巧。但是把 BN 看作一个层，有利于理解和程序实现。

7.4.7　将 BN 层应用于卷积网络

上面介绍的都是将 BN 层应用于全连接层，而将其应用于卷积网络时，本质和全连接层是一样的。4.2.5 节展示了通过对卷积网络的矩

阵化，卷积层变成了全连接层，即：

$$\begin{aligned}&\boldsymbol{U}_{\text{matric}}=\text{matric}(\boldsymbol{U})\\&\boldsymbol{X}=\boldsymbol{U}_{\text{matric}}\boldsymbol{W}\\&\boldsymbol{Y}=\text{BN}(\boldsymbol{X};\ \gamma,\beta)\\&\boldsymbol{Z}=f(\boldsymbol{Y})\end{aligned}\tag{7.7}$$

输入 4D 特征图 $\boldsymbol{U}$，矩阵化得到矩阵 $\boldsymbol{U}_{\text{matric}}$，后面和全连接层计算一模一样，即矩阵相乘，BN（批量规范化）和非线性激活。

上面通过公式说明了 BN 层如何应用于卷积网络，其物理意义是卷积网络采用了参数共享技术，所以希望一个输出 2D 特征图也要采用相同的规范化，即把它看作一个属性。变量 $\boldsymbol{X}$ 的列向量刚好是一个输出 2D 特征图（包含批量所有样本），所以对 $\boldsymbol{X}$ 进行批量规范化，即对 $\boldsymbol{X}$ 的列向量采用相同的规范化，能满足要求。总之，一个输出 2D 特征图只需一对(γ,β)参数，而不是一个神经元需要一对参数（如常规神经网络那样）。

7.5 数据扩增

深度学习的成功（如卷积网络）归功于 3 个方面：学习模型、大数据和强大的计算资源。其中大数据功不可没，甚至比学习模型还重要。在机器学习领域，只要拥有足够好的数据集，模型可以差些，所以在实践中建立好的训练集占用了大量精力。好的数据集要求样本数量巨大，尽可能覆盖整个分布空间。这是深度网络巨大的参数数量所要求的，如果训练数据过少，则很容易发生过拟合。著名物理学家费

米曾说："我记得我的朋友约翰 · 冯 · 诺依曼曾经说过，用 4 个参数可以拟合出一头大象，而用 5 个参数可以让它的鼻子摆动。" 这句名言是指：即使复杂模型很好地拟合了数据集，也不应该感到震惊，因为只要有足够的参数，就可以拟合任意数据集。

具体到图像分类任务，正如第 1 章所指出的，图像分类中面临的困难主要是由图像的几个不变性引起的：平移、旋转、镜像、尺度、色彩和形变等。比如，色彩不变性指物体不会因为颜色的改变而变为不同类物体。根据这些不变性，我们可以人为增加样本数量来减小过拟合，即通过图像处理技术来模拟这些不变性，使一个样本变为多个样本。

常用的数据扩增技术包括对图像进行平移、旋转、左右翻转来分别实现平移、旋转和镜像不变性；进行随机裁剪和缩放实现尺度不变性；进行仿射变换和弹性变形实现形变不变性；进行模糊、加噪、PCA Jittering 和 Color Jittering 实现色彩不变性。由于这些技术涉及很多图像处理知识并且工程化特征明显，本书不打算展开论述，读者可以查阅相关资料进行研究。

如果把样本看作高维空间的点，那么对样本做数据扩增，就是在样本点的"邻域"内生成新的样本点，达到增加样本数量的目的。"邻域"的概念在高维空间和低维空间不同，"邻域"不一定是指欧式距离近，而是"语义"距离近。由于数据扩增的样本点都位于"邻域"内，所以样本点云的覆盖范围并没有增大多少。在实践中，如果数据扩增后，模型的性能还没有达到要求，仍然必须通过采集新样本，使样本

点云的覆盖范围增大。

再次强调下，数据扩增技术一般只在模型参数数量巨大，而样本数量有限的情况下使用，特别是在深度学习中用来防止过拟合。同时要有有效的技术手段来实现数据扩增，比如图像中的不变性。

7.6 梯度检查

利用梯度下降法进行优化时，需要采用链式法则计算损失函数对参数的梯度，这是一个十分容易出错的地方，所以必须对梯度计算进行检查，以确保正确。梯度检查的总原则是比较数值梯度和解析梯度值是否一致，其中解析梯度值是采用链式法则计算的梯度值，数值梯度是采用梯度定义计算的梯度值。

假设我们需要对参数 w（标量）的梯度进行检查，损失函数为 $f(w)$，数值梯度定义：

$$f_n' = \frac{f(w+h)-f(w)}{h}$$

其中 h 是步长，是小常数，实践中常取 `10**(-5)`。计算数值梯度时，需要进行两次前向计算，效率十分低下。由于定义是一阶收敛，计算精度低，因此实际中采用中心差分法计算梯度：

$$f_n' = \frac{f(w+h)-f(w-h)}{2h}$$

这是二阶收敛，精度高。利用函数的泰勒展开式进行证明，如下：

$$f_n' = \frac{f(w+h)-f(w)}{h} = \frac{(f(w)+f'(w)h+O(h^2))-f(w)}{h} = f'(w)+O(h)$$

$$f(w+h) = f(w)+f'(w)h+0.5f''(w)h^2+O(h^3)$$

$$f(w-h) = f(w)-f'(w)h+0.5f''(w)h^2+O(h^3)$$

$$f_n' = \frac{f(w+h)-f(w-h)}{2h} = f'(w)+O(h^2)$$

下面列出一些注意事项。

- 使用很少的样本进行计算。当采用批量样本计算梯度时，批量应取 2 到 3，这样能保证计算精度更高，同时提高梯度检查效率。
- 注意正则化损失。当正则化（范数梯度很容易计算，不容易出错）损失比例较大时，梯度可能主要由正则化损失主导，数据损失即使有错，也不容易被发现。所以刚开始关闭正则化，只使用数据损失进行梯度检查，然后再慢慢增大正则化强度，同时检查正则化和数据损失。
- 关闭模型中的随机成分，如 dropout。
- 关闭程序中参数更新代码，计算数值梯度时，两次前向计算要采用相同的样本数据。
- 采用双精度浮点数进行计算。
- 对模型所有层参数进行梯度检查，注意权重和偏置要分开检查，因为偏置的梯度计算程序和权重不同。
- 样本数据最好采用标准正态随机数进行模拟，这样计算的梯度精度较高。
- 确保梯度值不能过分小，如果 f_n' 在 `10**(-5)` 量级以下，可以通过增大权重初始化时的方差或样本方差（采用标准正态随机数进行模拟时），从而使 f_n' 在 `10**(-3)` 量级。

- 采用相对误差来衡量一致性，假设 f_n' 是数值梯度，f_a' 是解析梯度，则采用指标：$\xi = |(f_n' - f_a')| / \max(|(f_n')|, |(f_a')|)$，当指标小于 `10**(-7)`时，说明梯度计算正确。
- 最后，偏置初始化为小的正常数（如 0.1），使 ReLU 处于激活状态。

7.7 初始损失值检查

梯度检查即使正确，也不能说明代码正确实现了模型，因为梯度检查只是验证了链式法则正确计算了损失函数的梯度而已。模型对不对，还需要进一步检查。最简单的检查方法是采用小的权重对模型进行初始化，关闭正则化，只检查数据损失，此时样本数据最好采用标准正态随机数进行模拟。例如，如果样本有 10 类，采用 softmax 损失函数，则初始数据损失应为 2.302。因为模型是随机初始化的，所以样本属于哪个类别的概率期望都是 1/10，则损失为 $-\log(1/10) = 2.302$。如果初始损失值不正确，则把权重的方差变小，如果仍不对，则可能是程序内部出错了。

7.8 过拟合微小数据子集

最后也是最重要的：在对整个训练集训练之前，首先对微小数据子集（比如每类 2 到 4 个样本）进行训练，争取获得很小的损失值（发生过拟合），此时也要关闭正则化。如果没有获得很小的损失值，则最

好不要对整个训练集进行训练。即使模型完全拟合了微小数据子集，也不能保证程序一定正确。因为模型相对于微小子集容量过大，所以当程序没有正确实现时，只要模型流程是正确的，梯度计算正确，模型就很容易过拟合微小子集。典型的错误有：忘了加激活层，最后一个全连接层后加了激活层，权重初始化不正确，权重更新错误，以及数据预处理错误等。

7.9　监测学习过程

要实时监控训练过程，监控训练过程的主要目的是观察超参数是否设置合理，学习效果如何以及是否发生了过拟合。比如学习率设置得过大，模型根本没有进行有效学习时，就需要及时终止训练，避免浪费计算资源。有两个最重要的超参数需要监控：学习率和正则化强度。

7.9.1　损失值

我们可以通过监测损失值随训练周期的变化情况来判断学习率的大小是否合适。有 4 种典型情况：第一种是损失值突然增大，大于初始损失值，而且持续增大，这说明学习率太大了；第二种是损失值减小速度很快或者波动太大，很快就饱和了，不再减小，但最终损失值过大，这说明学习率比较大；第三种是损失值减小速度很慢，这说明学习率过小；第四种是损失值减小速度适中，最终损失值很小，这说明学习率比较合适。实践中，前两种情况很容易判断，后两种情况不

好区分，只能采用不同的学习率进行训练，然后比较，选择合适的学习率。损失值一般采用批量的损失值，而不是整个训练集的损失值。由于批量的数量较小，所以损失值会有波动，当波动比较大时，可以提高批量的数量。损失值画图显示时，可以在对数域显示，特别是当损失值很小时，在对数域很容易观察到损失值的变化。

7.9.2 训练集和验证集的准确率

我们通过监测训练集和验证集的准确率差异随训练周期的变化情况，来判断正则化强度是否合适。有 3 种典型情况：第一种是验证集的准确率远差于训练集的准确率，这说明过拟合很明显；第二种是验证集的准确率稍差于训练集的准确率，这说明过拟合比较弱；第三种是验证集的准确率和训练集的准确率一致，这说明基本没有过拟合。当过拟合比较弱时，判断验证集的性能是否达到设计要求。如果达到要求，则此时模型可以作为最终模型；如果没有达到设计要求，可以增加模型容量，再进行训练，因为过拟合弱，可能是模型的学习容量小导致的。理论上计算训练集和验证集的性能需采用全部的样本，但这样计算代价太大。实践中，训练集的性能可以采用批量的性能，验证集的性能可以在验证集中随机选择批量数量的样本来计算。画图显示时，可以在对数域显示准确率，当准确率接近 100% 时，很容易观察到准确率的变化。

随着训练进行，当验证集的准确率开始变差时，说明发生了过拟合，可以提前终止训练。训练集的准确率一般来说，会越来越高，如果发生降低，则可能是程序出错。

7.9.3　参数更新比例

采用梯度下降法等方法更新参数时，通过监测参数的更新比例，可以掌握每层参数的学习情况。参数更新比例是计算一层中所有参数更新量和参数的比值，注意，需要每层单独监控。更新比例应该在 10**(-3)左右，如果大于此比例，可能学习率过大；如果小于此比例，学习率可能过小。参数更新比例代码如下：

```
update_param = -lr*dparam
update_ratio = np.sum(np.abs(update_param))
    /np.sum(np.abs(param))
```

其中 param 是一层的权重或偏置参数，dparam 是梯度，update_param 是参数更新量，update_ratio 是更新比例。

第 8 章

神经网络实例

本章采用一个模拟数据集进行神经网络的训练，把前面的相关知识整合在一起，包括数据预处理、BN 层、神经网络模型、梯度反向传播、梯度检查、监测训练过程、超参数随机搜索等，使读者掌握一个完整的机器学习流程。利用本章的程序，我们可以进行实际任务的训练和评估。本章采用结构化程序设计思想，把每个独立模块封装成函数，其中涉及的知识点在前面章节中都详细介绍过，而且程序本身也很容易理解，故不再进行详细解释。本章只是进行简单封装，直接给出相关代码，方便读者查阅。

8.1 生成数据

生成一个线性不可分的数据集,它是一个随时间增长的振荡数据。首先，设置数据集的一些参数：

```
num_samp_per_class = 200
dim = 2
N_class = 4
```

然后生成数据函数：

```
def gen_toy_data(dim, N_class, num_samp_per_class):
    num_examples = num_samp_per_class*N_class
    X = np.zeros((num_examples, dim))
    labels = np.zeros(num_examples, dtype = 'uint8')
    for j in range(N_class):
        ix = range(num_samp_per_class*j,num_samp_per_class*
            (j+1))
        x = np.linspace(-np.pi, np.pi, num_samp_per_class) + 5
        y = np.sin(x + j*np.pi/(0.5*N_class) )
        y += 0.2*np.sin(10*x + j*np.pi/(0.5*N_class) )
        y += 0.25*x + 10 # 线性增长
        y += np.random.randn(num_samp_per_class)*0.1 # 噪声

        X[ix] = np.c_[x, y]
        labels[ix] = j
    return (X,labels)
```

接着可视化数据，结果如图 8.1 所示。

```
import matplotlib.pyplot as plt
def show_data(X, labels):
    plt.scatter(X[:, 0], X[:, 1], c = labels, s = 40, cmap =
        plt.cm.Spectral)
    plt.show()
```

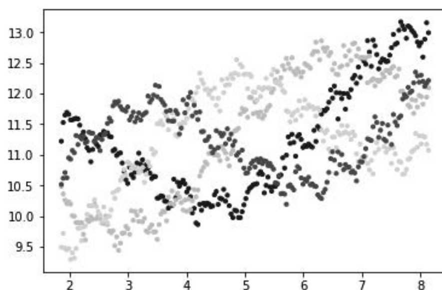


图 8.1　该数据集有 4 类样本，是线性增长的振荡数据（另见彩插）

8.2 数据预处理

中心化和归一化的代码如下：

```
def normalize(X):
    # (x-u)/delta
    mean = np.mean(X, axis = 0)
    X_norm = X - mean
    std = np.std(X_norm, axis = 0)
    X_norm / =  std + 10**(-5)
    return (X_norm, mean, std)
```

PCA 白化的代码如下：

```
def PCA_white(X):
    mean = np.mean(X, axis = 0)
    X_norm = X - mean
    cov = np.dot(X_norm.T, X_norm)/X_norm.shape[0]
    U,S,V = np.linalg.svd(cov)
    X_norm = np.dot(X_norm, U)
    X_norm /= np.sqrt(S + 10**(-5))
    return (X_norm, mean, U, S)
```

数据集按比例 2∶1∶1 随机分为训练集、验证集和测试集，相关代码如下：

```
def split_data(X, labels):
    num_examples = X.shape[0]
    shuffle_no = list(range(num_examples))
    np.random.shuffle(shuffle_no)

    X_train = X[shuffle_no[:num_examples//2]]
    labels_train = labels[shuffle_no[:num_examples//2]]

    X_val = X[shuffle_no[num_examples//2:num_examples//2 +
        num_examples//4]]
```

```
    labels_val = labels[shuffle_no[num_examples//2:num_
        examples//2 + num_examples//4]]

    X_test = X[shuffle_no[-num_examples//4 :]]
    labels_test = labels[shuffle_no[-num_examples//4 :]]

    return (X_train, labels_train, X_val, labels_val,
        X_test, labels_test)
```

然后对数据集进行预处理：

```
def data_preprocess(X_train, X_val, X_test):
    (X_train_pca, mean, U, S) = PCA_white(X_train)
    X_val_pca = np.dot(X_val-mean, U)
    X_val_pca /= np.sqrt(S + 10**(-5))
    X_test_pca = np.dot(X_test-mean, U)
    X_test_pca /= np.sqrt(S + 10**(-5))
    return (X_train_pca, X_val_pca, X_test_pca)
```

8.3　网络模型

神经网络模型的超参数主要是网络深度（隐含层的数量）和每层神经元的数量。我们可以采用 list 结构（`layer_param`）存储每层神经元数量，包括输入层和输出层。当 `layer_param` 只有两个元素时，表示没有隐含层，此时整个网络就是线性模型。模型的权重和偏置参数也可以采用 list 结构存储。

权重初始化代码如下：

```
# layer_param = [dim, 100, 100, N_class]
def initialize_parameters(layer_param):
    weights = []
```

```
        biases = []
        vweights = []
        vbiases = []

        for i in range(len(layer_param) - 1):
            in_depth = layer_param[i]
            out_depth = layer_param[i+1]
①           std = np.sqrt(2/in_depth)*0.5
            weights.append(std * np.random.randn(in_depth,
                out_depth))
            biases.append(np.zeros((1, out_depth)))
            vweights.append(np.zeros((in_depth, out_depth)))
            vbiases.append(np.zeros((1, out_depth)))
        return (weights, biases, vweights, vbiases)
```

其中语句 ① 中乘以 0.5 进行修正，使初始数据损失接近 `-log (1/N_class)`。

模型前向计算代码如下，这里需要注意最后一层不需要非线性激活：

```
def forward(X, layer_param, weights, biases):
    hiddens = []
    hiddens.append(X)
    for i in range(len(layer_param)-2 ):
        hiddens.append(np.maximum(0, np.dot(hiddens[i],
            weights[i]) + biases[i]) )
    scores = np.dot(hiddens[-1], weights[-1]) + biases[-1]
    return (hiddens, scores)
```

接下来列出相关的网络模型函数。

计算 softmax 数据损失值：

```
def data_loss_softmax(scores, labels):
    num_examples = scores.shape[0]
    exp_scores = np.exp(scores)
```

```
    exp_cores_sum = np.sum(exp_scores, axis = 1)
    corect_probs = exp_scores[range(num_examples),
        labels]/exp_cores_sum
    corect_logprobs =-np.log(corect_probs)
    data_loss = np.sum(corect_logprobs)/num_examples
    return data_loss
```

计算 L2 范数损失：

```
def reg_L2_loss(weights, reg):
    reg_loss = 0
    for weight in weights:
        reg_loss += 0.5*reg*np.sum(weight*weight)
    return reg_loss
```

计算分值矩阵梯度：

```
def dscores_softmax(scores, labels):
    num_examples = scores.shape[0]
    exp_scores = np.exp(scores)
    probs = exp_scores/np.sum(exp_scores, axis = 1,
        keepdims = True)
    dscores = probs
    dscores[range(num_examples), labels] -= 1
    dscores /= num_examples
    return dscores
```

准确率预测：

```
def predict(X, labels, layer_param, weights, biases):
    hidden = X
    for i in range(len(layer_param)-2):
        hidden = np.maximum(0, np.dot(hidden, weights[i]) +
        biases[i] )
    scores = np.dot(hidden, weights[-1])  + biases[-1]
    predicted_class = np.argmax(scores, axis = 1)
    rigth_class = predicted_class == labels
    return np.mean(rigth_class)
```

注：predict 函数和前向函数 forward 几乎一致，只是不需要保存 hidden 神经元。

梯度反向传播算法：

```
def gradient_backprop(dscores, hiddens, weights, biases, reg):
    dweights = []
    dbiases = []
    dhidden = dscores
    for i in range(len(hiddens)-1, -1, -1):
        dweights.append(np.dot(hiddens[i].T, dhidden) +
            reg*weights[i])
        dbiases.append(np.sum(dhidden , axis = 0,
            keepdims = True))
        dhidden = np.dot(dhidden, weights[i].T)
        dhidden[hiddens[i] <= 0] = 0
    return (dweights, dbiases)
```

8.4 梯度检查

对每层的权重和偏置进行梯度检查，数据集采用标准正态分布随机数，所以不需要预处理。相关代码如下：

```
def gen_random_data(dim, N_class, num_samp_per_class):
    num_examples = num_samp_per_class*N_class
    X = np.random.randn(num_examples, dim) # data matrix
    labels = np.random.randint(N_class, size = num_examples)
    return (X, labels)

def check_gradient(X, labels, layer_param,
    check_weight_or_bias):
#   (X, labels) = gen_random_data(dim, N_class,
    num_samp_per_class = 200)
```

```
#    layer_param = [dim, N_class]
#    layer_param = [dim, 10, 20, N_class]
#    check_weight_or_bias: 1 for weight, 0 for bias

     (weights, biases, vweights, vbiases) =
         initialize_parameters(layer_param)
     reg = 10**(-9)
     step = 10**(-5)
     for layer in range(len(weights)):
         if check_weight_or_bias:
             row = np.random.randint(weights[layer].shape[0])
             col = np.random.randint(weights[layer].shape[1])
①            param = weights[layer][row][col]
         else:
             row = np.random.randint(biases[layer].shape[1])
②            param = biases[layer][0][row]

         (hiddens, scores) = forward(X, layer_param,
             weights,biases)
         dscores = dscores_softmax(scores, labels)
         (dweights, dbiases) = gradient_backprop(dscores,
             hiddens, weights, biases, reg)
         if check_weight_or_bias:
③            danalytic = dweights[-1-layer][row][col]
         else:
④            danalytic = dbiases[-1-layer][0][row]

         if check_weight_or_bias:
             weights[layer][row][col] = param - step
         else:
             biases[layer][0][row] = param - step
         (hiddens, scores) = forward(X, layer_param,
             weights, biases)
         data_loss1 = data_loss_softmax(scores, labels)
         reg_loss1 = reg_L2_loss(weights, reg)
         loss1 = data_loss1 + reg_loss1

         if check_weight_or_bias:
```

```
        weights[layer][row][col] = param + step
    else:
        biases[layer][0][row] = param + step
    (hiddens, scores) = forward(X, layer_param,
        weights, biases)
    data_loss2 = data_loss_softmax(scores, labels)
    reg_loss2 = reg_L2_loss(weights, reg)
    loss2 = data_loss2 + reg_loss2
    dnumeric = (loss2 - loss1)/(2*step)

    print(layer, data_loss1, data_loss2)
    error_relative = np.abs(danalytic - dnumeric)/
        np.maximum(danalytic, dnumeric)
    print(danalytic, dnumeric, error_relative)
```

在上述代码中，语句①和语句②取出一层的任一权重或偏置进行测试。这里需要注意语句③和语句④中的`-1-layer`。因为梯度`dweights`是倒序存储的，即模型第一层的梯度存储在列表最后。初始数据损失在 1.38 左右，和理论值`-np.log(1/N_class)`十分接近。梯度值`danalytic`和`dnumeric`数值变化很大，在`10**(-2)`到`10**(-4)`之间。相对误差`error_relative`在`10**(-8)`量级以下，这说明梯度检查正确。

8.5 参数优化

这里我们采用 Nesterov 动量优化方法，同时函数返回了参数更新比例：

```
def nesterov_momentumSGD(vparams, params, dparams, lr, mu):
    update_ratio = []
    for i in range(len(params)):
        pre_vparam = vparams[i]
```

```
            vparams[i] = mu*vparams[i] - lr*dparams[-1-i]
            updata_param = vparams[i] + mu*(vparams[i] -
                pre_vparam)
            params[i] += updata_param
            update_ratio.append(np.sum(np.abs(updata_param))
                /np.sum(np.abs(params[i])))
        return update_ratio
```

8.6　训练网络

训练网络的代码如下：

```
def train_net(X_train, labels_train, layer_param, lr,
    lr_decay, reg, mu, max_epoch,  X_val, labels_val):

①       (weights, biases, vweights, vbiases) =
            initialize_parameters(layer_param)
    epoch = 0
    data_losses = []
    reg_losses = []

    val_accuracy = []
    train_accuracy = []
    weights_update_ratio = []
    baises_update_ratio = []
    while epoch < max_epoch:
②       (hiddens, scores) = forward(X_train, layer_param,
            weights, biases)

③       val_accuracy.append(predict(X_val, labels_val, l
            ayer_param, weights, biases))
        train_accuracy.append(predict(X_train, labels_train,
            layer_param, weights, biases))

④       data_loss = data_loss_softmax(scores, labels_train)
        reg_loss = reg_L2_loss(weights, reg)
```

```
⑤       dscores = dscores_softmax(scores, labels_train)
⑥       (dweights, dbiases) = gradient_backprop(dscores,
            hiddens, weights, biases, reg)
⑦       weights_update_ratio.append( nesterov_momentumSGD
            (vweights, weights, dweights, lr, mu) )
        baises_update_ratio.append( nesterov_momentumSGD
            (vbiases, biases, dbiases, lr, mu) )
        data_losses.append(data_loss)
        reg_losses.append(reg_loss)
        epoch += 1
⑧       lr *= lr_decay

    # 可视化数据损失、训练集和验证集的准确率
    plt.close()
    fig = plt.figure('loss')
    ax = fig.add_subplot(2,1,1)
    ax.grid(True)
    ax2 = fig.add_subplot(2,1,2)
    ax2.grid(True)
    plt.xlabel( 'log10(lr) = ' + str(round((np.log10(lr)),
        2)) + ' ' +  'log10(reg) = ' + str(round((np.log10
        (reg)),2)), fontsize = 14)
    plt.ylabel('accuracy log10(data
        loss)', fontsize = 14)
    ax.scatter(np.arange(len(data_losses)), np.log10(data_
        losses), c = 'b',marker = '.')
#   ax2.scatter(np.arange(len(reg_losses)),
        np.log10(reg_losses), c = 'r',marker = '*')
    ax2.scatter(np.arange(len(val_accuracy)-0),
        val_accuracy[0:], c = 'r',marker = '*')
    ax2.scatter(np.arange(len(val_accuracy)-0),
        train_accuracy[0:], c = 'g',marker = '.')
    plt.show()

    # 对数显示每层权重和偏置的更新率，合理值在10**(-3)
    for layer in range(len(weights)):
        wur = []
        for i in range(len(weights_update_ratio)):
```

```
            wur.append( weights_update_ratio[i][layer] )

        bur = []
        for i in range(len(baises_update_ratio)):
            bur.append( baises_update_ratio[i][layer] )

        plt.close()
        fig = plt.figure('update ratio')
        ax = fig.add_subplot(2,1,1)
        ax.grid(True)
        ax2 = fig.add_subplot(2,1,2)
        ax2.grid(True)
        plt.xlabel( 'log10(lr) = ' + str(round((np.log10
            (lr)),2)) + ' ' +  'log10(reg) = ' + str(round
            ((np.log10(reg)),2)), fontsize = 14)
        ax.scatter(np.arange(len(wur)), np.log10(wur),
            c = 'b',marker = '.')
        ax2.scatter(np.arange(len(bur)), np.log10(bur),
            c = 'r',marker = '*')
        plt.show()

    return (data_losses, reg_losses, weights, biases,
        val_accuracy)
```

这里简要介绍一下训练网络的流程。首先，初始化参数（语句①）。接着，前向计算得到分值矩阵和隐含层激活值（语句②），计算训练集和验证集的准确率（语句③），计算数据损失和正则化损失（语句④）。然后，开始梯度反向传播。先计算分值矩阵的梯度（语句⑤），然后进行反向传播（语句⑥），计算权重和偏置更新率（语句⑦）。接着，进行学习率指数退火（语句⑧）。最后，对损失、准确率及参数更新率等进行可视化。

8.7 过拟合小数据集

过拟合小数据集的相关代码如下：

```
def overfit_tinydata(X, labels, layer_param, lr = 10**(-0.0),
    lr_decay = 1, mu = 0.9, reg = 0, max_epoch = 100):
#   (X,labels) = gen_toy_data(dim, N_class,
    num_samp_per_class = 2)
#   layer_param = [dim, 100, 100, N_class]
    (data_losses, reg_losses, weights, biases, accuracy) =
        train_net(X, labels, layer_param, lr, lr_decay, reg,
        mu, max_epoch, X, labels)

    return (data_losses, reg_losses, accuracy)
    # data_loss = 5.932608313857891e-04
```

此时最好关闭正则化（reg = 0），数据集必须非常小，本实例使用了每类两个样本。训练了 100 轮，最终数据损失是 5.9e-04，准确率为 100%，这说明已经发生过拟合。采用的模型有两个隐含层，每层都有 100 个神经元。

8.8 超参数随机搜索

超参数随机搜索的相关代码如下：

```
def hyperparam_random_search(X_train, labels_train, X_val,
    labels_val, layer_param, num_try = 10, lr = [-1, -5],
    lr_decay = 1, mu = 0.9, reg = [-2.0, -5.0], max_epoch = 500):
#   (X,labels) = gen_toy_data(dim, N_class,
    num_samp_per_class = 200)
#   layer_param = [dim, 100, 100, N_class]
```

```
minlr = min(lr)
maxlr = max(lr)
randn = np.random.rand(num_try*2)
lr_array = 10**(minlr + (maxlr - minlr)*randn[0: num_try])
minreg = min(reg)
maxreg = max(reg)
reg_array = 10**(minreg + (maxreg - minreg)*randn[num_try:
    2*num_try])
lr_regs =  zip(lr_array, reg_array)

for lr_reg in lr_regs:
    (data_loss, reg_loss, weights, biases, val_accuracy)
        = train_net(X_train, labels_train, layer_param,
        lr_reg[0], lr_decay, lr_reg[1], mu, max_epoch,
        X_val, labels_val)

return (weights, biases)
```

注意，调用该函数时，必须使用 8.2 节介绍的预处理后的数据集。由于数据集很小，收敛速度较快，所以学习率不需要退火（lr_decay = 1）。采用模型 layer_param = [dim = 2, 100, 100, N_class = 4] 进行实际训练时，会得到一些结果，如图 8.2 所示。每个结果都有两组图：左上图是数据损失（对数显示），左下图是验证集和训练集的准确率（验证集曲线位于下面）；右上图是第一层权重更新比例，右下图是第一层偏置更新比例，注意比例都是对数显示的。

首先，进行超参数粗搜索，进行 500 轮训练，结果如图 8.2 所示，表明学习率 10**(-0.1)过大。

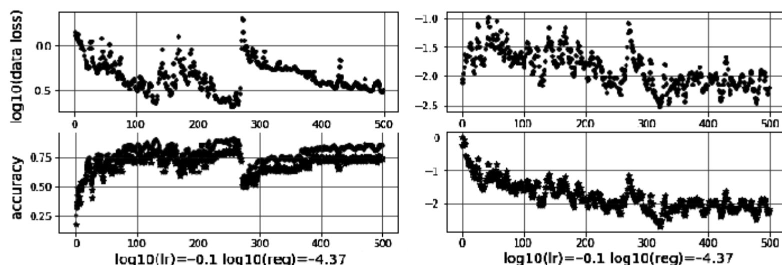


图 8.2　学习率过大，数据损失波动很大且下降很快，参数更新率也过高（另见彩插）

接着减小学习率，进行 2000 轮训练，结果如图 8.3 所示，这表明学习率 `10**(-1.18)`稍微偏大。

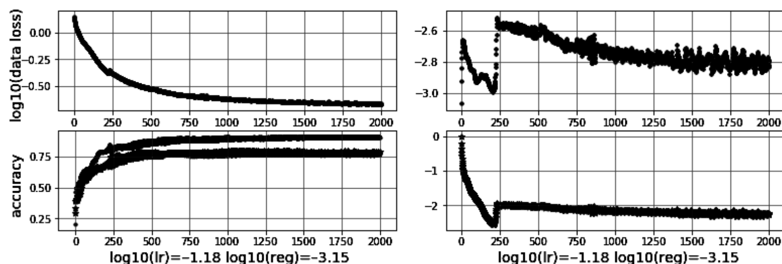


图 8.3　数据损失无波动，训练集和验证集的准确率饱和，验证集的准确率只有 75%，参数更新率稍大，这说明减小学习率有可能进一步提高验证集的准确率（另见彩插）

继续减小学习率，进行 10 000 轮训练，结果如图 8.4 所示，这表明学习率 `10**(-2.1)`比较合适。

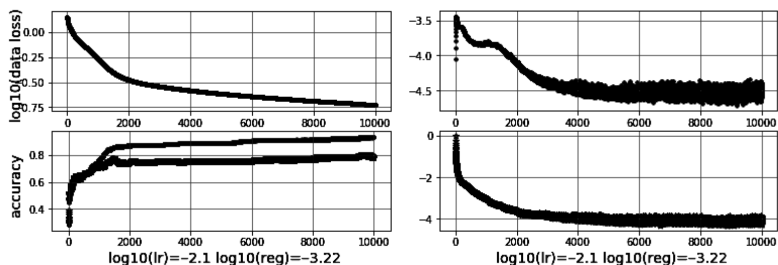


图 8.4　数据损失无波动，训练集和验证集的准确率趋于饱和，验证集的准确率有 80%，训练集的准确率高达 95%，收敛时参数更新率已经很低，过拟合现象不明显，这说明模型验证集的准确率难以超过 80%（另见彩插）

8.9　评估模型

程序的最后，采用测试集对模型进行评估，得到模型的最终泛化性能指标，其参数使用上节验证集中准确率最高的模型参数。相关代码如下：

```
accuracy = predict(X_test_pca, labels_test,
    layer_param, weights, biases)
```

8.10　程序组织结构

整个程序有很多函数，如果全部放在一个文件中，那么结构会不清晰，不利于扩展。经过分析可以发现，程序主要由三部分组成：数据集处理函数（8.1 节和 8.2 节）、网络模型函数（8.3 节、8.5 节和 8.6 节）和模型使用函数（8.4 节、8.7 节到 8.9 节）。将这三部分函数分别存储为

一个文件，作为一个独立模块，其文件名分别为 data_process、nn_model 和 nn_model_utilize，其中 nn_model_utilize 模块需导入 nn_model 模块。

主文件nn_test调用nn_model_utilize模块的函数,代码如下：

```
from data_process import *
from nn_model_utilize import *

if __name__ == '__main__':
    #%%
    dim = 2 # dimensionality
    N_class = 4 # number of classes

    #%%
    layer_param = [dim, 10, 20, N_class]
    (X, labels) = gen_random_data(dim, N_class,
    num_samp_per_class = 20)
    for i in range(2):
        check_gradient(X, labels, layer_param, 1)
#   #%%
    layer_param = [dim, 100, 100, N_class]
    (X,labels) = gen_toy_data(dim, N_class,
        num_samp_per_class = 2)
    X,_,_,_ = PCA_white(X)
    (data_losses, reg_losses, accuracy) = overfit_tinydata
       (X, labels, layer_param, lr = 10**(-0.5), lr_decay = 1,
        mu = 0.9, reg = 10**(-10), max_epoch = 100)
#   #%%
    layer_param = [dim, 100, 100, N_class]
    (X,labels) = gen_toy_data(dim, N_class,
        num_samp_per_class = 200)
    (X_train, labels_train, X_val, labels_val, X_test,
        labels_test) = split_data(X, labels)
    (X_train_pca, X_val_pca, X_test_pca) =
        data_preprocess(X_train, X_val, X_test)
    (weights, biases) = hyperparam_random_search(X_train_
```

```
pca, labels_train, X_val_pca, labels_val, layer_param,
                            num_try = 2, lr = [-1, -1.01],
                            lr_decay = 1, mu = 0.9,
                            reg = [-2, -5], max_epoch = 200)
```

8.11　增加 BN 层

增加 BN 层，不会改变程序的总体流程。但是需要注意的是，这里不再需要偏置参数，而是增加了标准差和均值参数，同时需要保留 BN 层的一些中间结果，如均值和方差等。注意，由于采用全部样本进行训练，所以均值和方差不需要进行移动平均。

1. BN 层前向和后向代码

BN 层的前向计算代码如下：

```
BN_EPSILON = 10**(-5)
def BN_forward(X, gamma, beta):
    mu = np.mean(X, axis = 0)
    var = np.var(X, axis = 0)
    std = np.sqrt(var + BN_EPSILON)
    X_hat = (X - mu)/std
    Y = gamma*X_hat + beta
    bn_cache = (mu, var, std)
    return (Y, bn_cache, X_hat)
```

BN 层的后向计算代码如下：

```
def BN_backprop(dY, X, X_hat, bn_cache, gamma, beta):
    (mu, var, std) = bn_cache
    batch = X.shape[0]
    dX_hat = gamma*dY
    dbeta = np.sum(dY, axis = 0, keepdims = True)
    dgamma = np.sum(X_hat*dY, axis = 0, keepdims = True)
```

```
    dX = dX_hat/std
    dmu = -np.sum(dX_hat, axis = 0, keepdims = True)/std
    dstd = -1/(std*std) * np.sum((X - mu)*dX_hat, axis = 0)
    dvar = 1/2*(var**(-0.5))*dstd
    dX += 2*(X - mu)/batch * dvar
    dX += dmu/batch

    return (dX, dgamma, dbeta)
```

2. 模型参数的初始化和前向计算

参数初始化时，需要注意将方差参数 gammas 初始化为 1，而权重参数不再需要修正：

```
def initialize_parameters_BN(layer_param):
    weights = []
    gammas = []
    betas = []
    vweights = []
    vgammas = []
    vbetas = []

    for i in range(len(layer_param) - 1):
        in_depth = layer_param[i]
        out_depth = layer_param[i+1]
        std = np.sqrt(2/in_depth)
        weights.append( std * np.random.randn(in_depth,
            out_depth) )
        gammas.append( np.ones((1, out_depth)) )
        betas.append( np.zeros((1, out_depth)) )

        vweights.append( np.zeros( (in_depth, out_depth) ) )
        vgammas.append( np.zeros((1, out_depth)) )
        vbetas.append( np.zeros((1, out_depth)) )

    params = (weights, gammas, betas)
```

```
        vparams = (vweights, vgammas, vbetas)
        return (params, vparams)
```

网络前向计算的流程和没有 BN 层时一致，即先是全连接层（语句 ①），然后是 BN 层（语句 ②），接着进行非线性激活（语句 ③）。这里需要注意分值不需要激活（语句 ④）。相关代码如下：

```
    def forward_BN(X, layer_param, params):
        hiddens = []
        bn_ins = []
        bn_caches = []
        bn_hats = []

        (weights, gammas, betas) = params
        hiddens.append(X)
        for i in range(len(layer_param)-1):
①           hidden = np.dot(hiddens[i], weights[i])
                bn_ins.append(hidden)
②           (hidden, bn_cache, bn_hat) = BN_forward(hidden,
                gammas[i],betas[i])
            bn_hats.append(bn_hat)
            bn_caches.append(bn_cache)
            if i < len(layer_param)-2:
③               hiddens.append(np.maximum(0, hidden))
            else:
④               scores = hidden
        return (hiddens, scores, bn_ins, bn_hats, bn_caches)
```

3. 模型梯度反向传播

模型梯度反向传播代码如下：

```
    def gradient_backprop_BN(dscores, params, hiddens, bn_ins,
        bn_hats, bn_caches, reg):
        (weights, gammas, betas) = params
        dweights = []
```

```
    dgammas = []
    dbetas = []

    dhidden = dscores
    for i in range(len(hiddens)-1, -1, -1):
        (dhidden, dgamma, dbeta) = BN_backprop(dhidden,
        bn_ins[i], bn_hats[i], bn_caches[i], gammas[i],
        betas[i])
        dgammas.append(dgamma)
        dbetas.append(dbeta)
        dweights.append(np.dot(hiddens[i].T, dhidden) +
        reg*weights[i] )
        dhidden = np.dot(dhidden, weights[i].T)
        dhidden[hiddens[i] <= 0] = 0
    return (dweights, dgammas, dbetas)
```

其他函数代码与不含 BN 层的神经网络代码一致，这里不再贴出。

4. 训练结果分析

这里数据的初始损失值不再是-np.log(1/N_class)，变动范围比较大。梯度的相对误差也比较大，主要在 10**(-5)左右。BN 层的最大作用是采用更大的学习率以加速训练以及采用较小的正则化强度。这些结论得到了下面实验结果的支持。虽然初始学习率很大（10**(0.07)），正则化趋近 0（10**(-40)），数据损失波动较大且下降很快（200 轮训练），参数更新率也比较大，但是最终的验证集的准确率却达到 80%。这和上面没有 BN 层时的最好结果一致，如图 8.5 所示。

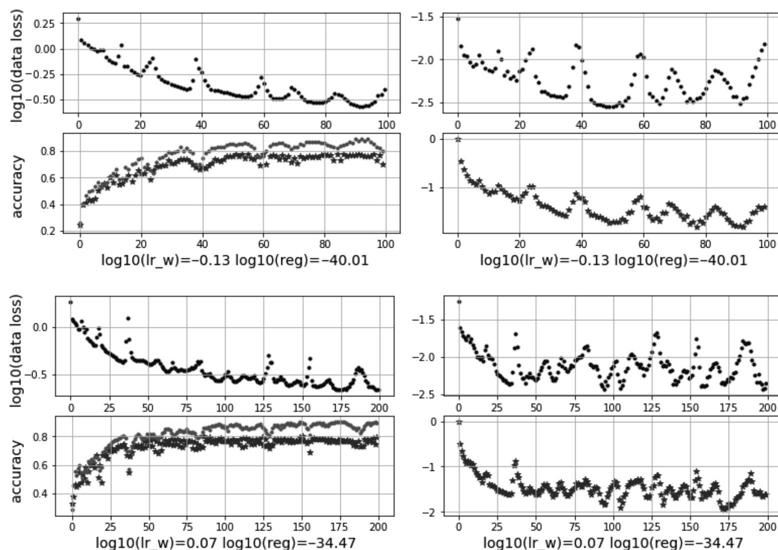


图 8.5　很大的初始学习率和很小的正则化，达到很好的性能（另见彩插）

8.12　程序使用建议

如果读者不对神经网络模型进行改进，那么使用本程序时只需改变网络超参数，如设置 `layer_param = [dim, 200, 100, 100, 100, N_class]` 和使用自己的数据矩阵及数据预处理，然后直接调用超参数随机搜索函数进行训练，保存最好模型的参数即可。如果对神经网络进行改进，只需要修改 8.3 节中的参数初始化、前向计算以及梯度反向传播算法，其他模块无须修改，具体可参考增加 BN 层的代码。由此可见，在实际工作中，精力主要集中在数据预处理和模型超参数搜索中，对模型的改进只是占用了很少的精力，甚至不需要改进模型，可直接使用基本模型。

第 9 章

卷积神经网络实例

第 8 章介绍了常规神经网络方面的例子，本章将采用 MNIST 数据集进行卷积神经网络实战学习,它与常规神经网络的流程基本一致，包括数据预处理、网络模型、梯度反向传播、梯度检查、监测训练过程和超参数随机搜索等知识。由于卷积神经网络的代码比较复杂，采用结构化设计会比较困难，所以本章采用面向对象的设计方法，把每个独立模块封装成对象。和第 8 章一样，每个知识点在前面章节中都详细介绍过，而且程序本身也很容易理解，故不再进行详细解释，只是进行简单的封装，给出接口。

本章的网络结构是类 VGG 结构，即 4.5 节介绍的最基本结构：将卷积层、池化层和全连接层这三层进行简单串联。程序的结构清晰，容易学习，和常规神经网络的实现十分接近，而且实际应用也较为广泛。

9.1　程序结构设计

训练一个卷积网络，主要包括 7 部分：激活函数、正则化、优化方法、卷积网络基本模块、训练方法、网络结构和数据集。每个部分都可以抽象成一个类，其中激活函数、正则化、优化方法、卷积网络基本模块和训练方法这 5 个类基本固定，可以适用于各种网络结构，并且前 4 个类都设计为接口类，数据存储在网络结构类中。网络结构类利用卷积网络的基本模块类进行组合，可以实现各种网络结构，如 ResNet。数据集类主要进行数据集的载入和预处理。9.2 节 ~ 9.8 节依次介绍了这 7 部分。

9.2　激活函数

该类实现了最常用的两种激活函数 ReLU 和 ELU。接口类没有数据，成员函数采用静态方法（@staticmethod），并对输入进行合理性检查（check_activation），相关代码如下：

```
import numpy as np

class ActivationInterface(object):
    activations = ['ReLU', 'ELU']
    @staticmethod
    def activation(data, activation):
        if activation == 'ReLU':
            data = np.maximum(0, data)
            return data
        if activation == 'ELU':
            expdata = np.exp(data) - 1
```

```
            data = np.where(data > 0, data, expdata)
            return data
    @staticmethod
    def dactivation(ddata, data, activation):
        if activation == 'ReLU':
            ddata[data <= 0] = 0
            return ddata
        if activation == 'ELU':
            ddatatemp = ddata*(data+1)
            ddata = np.where(data > 0, ddata, ddatatemp)
            return ddata
    @staticmethod
    def check_activation(activation):
        if activation not in ActivationInterface.activations:
            raise ValueError('''Activation methods: ReLU,
                ELU!''')
```

9.3 正则化

该类实现了两种范数正则化：L2 和 L1。相关代码如下：

```
import numpy as np

class RegulationInterface(object):
    regulations = ['L1', 'L2']
    @staticmethod
    def norm_reg(weight, reg, regulation):
        if regulation == 'L2':
            return np.sum(weight*weight)*reg/2
        if regulation == 'L1':
            return np.sum(np.abs(weight))*reg
    @staticmethod
    def dnorm_reg(weight, reg, regulation):
        if regulation == 'L2':
            return weight*reg
        if regulation == 'L1':
```

```
            return np.sign(weight)*reg
    @staticmethod
    def check_regulation(regulation):
        if regulation not in RegulationInterface.regulations:
            raise ValueError('''Regulation methods: L1,
                L2!''')
```

9.4　优化方法

OptimizerInterface 类实现了最常用的两种优化方法：Nesterov 动量方法和 Adam 自适应方法。其代码如下：

```
import numpy as np

class OptimizerInterface(object):
    optimizers = [ 'Nesterov', 'adam']
    decay_rate = 0.999
    eps = 10**(-8)
    @staticmethod
    def nesterov_momentumGD(lr, param, vparam, dparam,
        mu = 0.9):
        pre_vparam = vparam
        vparam = mu*vparam - lr*dparam
        updata_param = vparam + mu*(vparam - pre_vparam)
        update_ratio = np.sum(np.abs(updata_param))/
            np.sum(np.abs(param))
        param += updata_param
        return update_ratio
    @staticmethod
    def adam(lr, param, vparam, cache, dparam, t = 1,
        mu = 0.9):
        vparam = mu*vparam + (1-mu)*dparam
        vparamt = vparam/(1 - mu**t)
        cache = OptimizerInterface.decay_rate*cache +
            (1-OptimizerInterface.decay_rate)*(dparam**2)
```

```
        cachet = cache/(1 - OptimizerInterface.decay_rate**t)
        updata_param =-(lr/(np.sqrt(cachet) +
            OptimizerInterface.eps)) * vparamt
        update_ratio = np.sum(np.abs(updata_param))/
            np.sum(np.abs(param))
        param += updata_param
        return update_ratio

    @staticmethod
    def check_optimizer(optimizer):
        if optimizer not in OptimizerInterface.optimizers:
            raise ValueError('''updates methods: Nesterov
                and adam!''')
```

9.5 卷积网络的基本模块

下面给出了卷积层、池化层、全连接层、softmax 层的前向和反向，以及参数初始化代码。代码在第 4 章中有过详细解释，本节不再重复。注意，该类继承了激活类，因为需要实现激活函数。

```
import numpy as np

from activation_interface import ActivationInterface

class CnnBlockInterface(ActivationInterface):
    '''
    the implementation of three basic blocks of cnn net:
    the conv pool and fc block
    and the softmax layer
    '''
    @staticmethod
    def conv_layer(in_data, weights, biases, layer_param =
        (0,3,1,1), activation = 'ReLU'):
        '''
```

```
        in_data.shape = [batch,in_height,in_width,in_depth]
        weights.shape = [filter_size*filter_size*in_depth,
            out_depth]
        biases.shape = [1, out_depth]

        out_data.shape = [batch,out_height,out_width,
            out_depth]
        the data for calu gradient: matric_data, filter_data
        '''
        return (matric_data, filter_data, out_data)

    @staticmethod
    def dconv_layer(dout_data, matric_data, filter_data,
        weights, maps_shape, layer_param = (3,1,1),
        activation = 'ReLU'):
        '''
        inputs: dout_data, matric_data, filter_data
        matric_data, filter_data are data produced in the
            forward
        outputs: (dweight, dbias, din_data)
        '''
        return (dweight, dbias, din_data)

    @staticmethod
    def pooling_layer(in_data, filter_size = 2, stride = 2):
        '''
        in_data.shape = [batch,in_height,in_width,in_depth]
        out_data.shape = [batch,out_height,out_width,
            out_depth = in_depth]
        the data for calu gradient: matric_data_max_pos
        '''
        return (out_data, matric_data_max_pos)

    @staticmethod
    def dpooling_layer(dout_data, matric_data_max_pos,
        maps_shape, filter_size = 2, stride = 2):
        '''
        dout_data.shape = [batch,out_height,out_width,
```

```
            out_depth = in_depth]
        matric_data_max_pos.shape = [batch,in_height,
            in_width,in_depth]

        din_data.shape = [batch,in_height,in_width,in_depth]
        '''
        pass
        return din_data

    @staticmethod
    def FC_layer(in_data, weights, biases, out_depth, last,
        activation = 'ReLU'):
        '''
        in_data.shape = [batch, in_height, in_width, in_depth]
        weights.shape = [filter_size*filter_size*in_depth,
            out_depth]
        biases.shape = [1, out_depth]
        last = 1 if the FC is the last one

        out_data.shape = [batch,out_height,out_width,out_depth]
        the data for calu gradient: matric_data, filter_data
        '''
        (batch, in_height, in_width, in_depth) = in_data.shape
        matric_data = np.zeros( (batch, in_height*in_width*
            in_depth) )
        for i_batch in range(batch):
            matric_data[i_batch] = in_data[i_batch].ravel()
        filter_data = np.dot(matric_data, weights) + biases
        if not last: # 最后一层不需要激活
            filter_data = CnnBlockInterface.activation
                (filter_data, activation)

        out_data = np.zeros((batch, 1, 1, out_depth))
        for i_batch in range(batch):
            out_data[i_batch] = filter_data[i_batch]

        return (matric_data, filter_data, out_data)
```

```
@staticmethod
def dFC_layer(dout_data, matric_data, filter_data,
    weights, maps_shape, last, activation = 'ReLU'):
    '''
    inputs: dout_data, matric_data, filter_data
    matric_data, filter_data are data produced in the forward
    outputs: (dweight, dbias, din_data)
    '''
    (in_height, in_width, in_depth) = maps_shape
    (batch, out_height, out_width, out_depth) =
        dout_data.shape

    dfilter_data = np.zeros_like(filter_data)

    for i_batch in range(batch):
        dfilter_data[i_batch] = dout_data[i_batch].ravel()
    if not last:
        dfilter_data = CnnBlockInterface.dactivation
            (dfilter_data, filter_data, activation)

    dweight = np.dot(matric_data.T, dfilter_data)
    dbias = np.sum(dfilter_data, axis = 0, keepdims = True)
    dmatric_data = np.dot(dfilter_data, weights.T)

    din_data = np.zeros((batch, in_height, in_width,
        in_depth) )
    for i_batch in range(batch):
        din_data[i_batch] = dmatric_data[i_batch].reshape
            (in_height, in_width, -1)

    return (dweight, dbias, din_data)

@staticmethod
def softmax_layer(scores):
    """
    scores.shape = [batch,1,1,in_depth]
    probs.shape = [batch,1,1,in_depth]
    """
```

```
        scores -=  np.max(scores, axis = 3, keepdims = True)
        exp_scores = np.exp(scores)+10**(-8) # 数值计算更稳定
        exp_scores_sum = np.sum(exp_scores, axis = 3,
            keepdims = True)
        probs = exp_scores/exp_scores_sum
        return probs

    @staticmethod
    def data_loss(probs, labels):
        """
        labels is array of integers specifying correct class
        probs.shape = [batch,1,1,in_depth]
        """
        probs_correct = probs[range(probs.shape[0]), :, :,
            labels]
        logprobs_correct = -np.log(probs_correct)
        data_loss = np.sum(logprobs_correct)/labels.
            shape[0]
        return data_loss

    @staticmethod
    def evaluate_dscores(probs, labels):
        '''
        probs.shape = [batch,1,1,in_depth]
        labels is array of integers specifying correct class
        dscores.shape = [batch,1,1,in_depth]
        '''
        dscores = probs.copy()
        dscores[range(probs.shape[0]), :, :, labels] -= 1
        dscores /= labels.shape[0]
        return dscores

    @staticmethod
    def param_init(out_depth, in_depth, filter_size2):
        '''
        filter_size2 = filter_size*filter_size
        weights.shape = [filter_size2*in_depth, out_depth]
        '''
```

```
        std = np.sqrt(2)/np.sqrt(filter_size2*in_depth)
        weights = std * np.random.randn(filter_size2*
            in_depth, out_depth)
        biases = np.zeros((1, out_depth))
        return (weights, biases)
```

9.6　训练方法

CnnTrainInterface 类的代码如下：

```
import numpy as np
import matplotlib.pyplot as plt

class CnnTrainInterface(object):
    '''
    decay the learning rate every epoch using an exponential
        rate of lr_decay
    support learning rate and regularization random search
    also support train and test from checkpoint
    '''
    def __shuffle_data(self):
        shuffle_no = list(range(self.num_train_samples))
        np.random.shuffle(shuffle_no)
        self.train_labels = self.train_labels[shuffle_no]
        self.train_data = self.train_data[shuffle_no]

        shuffle_no = list(range(self.num_val_samples))
        np.random.shuffle(shuffle_no)
        self.val_labels = self.val_labels[shuffle_no]
        self.val_data = self.val_data[shuffle_no]

    def __train(self, epoch_more = 20, lr = 10**(-4), reg =
        10**(-5),  batch = 64, lr_decay = 0.8, mu = 0.9,
        optimizer = 'Nesterov', regulation = 'L2',
            activation = 'ReLU'):
        # 可视化数据损失、训练集和验证集准确率
```

```
        plt.close()
        fig = plt.figure('')
        ax = fig.add_subplot(3,1,1)
        ax.grid(True)
        ax2 = fig.add_subplot(3,1,2)
        ax2.grid(True)
        ax3 = fig.add_subplot(3,1,3)
        ax3.grid(True)
        plt.xlabel( 'log10(lr) = ' + str(round((np.log10(lr)),
            2)) + ' ' +  'log10(reg) = ' + str(round((np.log10
          (reg)),2)), fontsize = 14)
        plt.ylabel('update_ratio
            accuracy log10(data loss)', fontsize = 14)

        epoch = 0
        val_no = 0
        per_epoch_time = self.num_train_samples//batch
        while epoch < epoch_more:
            losses = 0
            self.__shuffle_data()
            for i in range(0, self.num_train_samples, batch):
                batch_data = self.train_data[i:i+batch,:]
                labels = self.train_labels[i:i+batch]
                (data_loss, reg_loss) = self.forward
                    (batch_data, labels, reg, regulation,
                    activation)
                losses += data_loss + reg_loss
                self.backpropagation(labels, reg,
                    regulation, activation)
                self.params_update(lr, per_epoch_time*
                    epoch + i+1, mu, optimizer)
                update_ratio = self.update_ratio[0][0]

            if i % (batch*20) == 0:
                ax.scatter(i/self.num_train_samples+epoch,
                    np.log10(data_loss), c = 'b',marker = '.')
                train_accuracy = self.predict(batch_data,
                    labels, activation)
```

```
        batch_data_val = self.val_data[val_no:
            val_no+batch,:]
        labels_val = self.val_labels[val_no:
            val_no+batch]
        val_accuracy = self.predict(batch_data_
            val,labels_val, activation)
        val_no += batch
        if val_no >=  self.num_val_samples - batch:
            val_no = 0
        ax2.scatter(i/self.num_train_samples+epoch,
            (train_accuracy), c = 'r',marker = '*')
        ax2.scatter(i/self.num_train_samples+epoch,
            (val_accuracy), c = 'b',marker = '.')
        ax3.scatter(i/self.num_train_samples+epoch,
            np.log10(update_ratio), c = 'r',marker = '.')
        plt.pause(0.000001)
    epoch += 1

    plt.savefig('checkpoint_' + '(loss_' + str(round
        (np.log10(losses/per_epoch_time),2)) +
            ')_(epoch_' +  str(round(epoch,2)) +
                ')_' + '_[(lr reg)_' + '(' + str
                (round((np.log10(lr)),2)) +
                ' ' + str(round((np.log10(reg)),2))
                + ')]' + '_' + ' ' + optimizer +
                ' '+ regulation + ' ' +
                activation + '.png')

    self.context[0] = lr
    self.save_checkpoint('checkpoint_' + '(loss_' +
        str(round(np.log10(losses/per_epoch_time),
        2)) + ')_(epoch_' + str(round(epoch,2)) +
        ')_' + '_[(lr reg)_' + '(' + str(round
        ((np.log10(lr)),2)) + ' ' + str(round
        ((np.log10(reg)),2)) + ')]' + '_' +
        ' ' + optimizer + ' '+ regulation + ' ' +
        activation + '.npy')
```

```
        lr * =  lr_decay # 使用指数衰减进行学习率退火

    self.test(batch, activation)

def __methods_check(self, optimizer, regulation,
    activation):
    self.check_optimizer(optimizer)
    self.check_regulation(regulation)
    self.check_activation(activation)

@staticmethod
def __gen_lr_reg(lr = [0, -6], reg = [-3, -6], num_try = 10):
    minlr = min(lr)
    maxlr = max(lr)
    randn = np.random.rand(num_try*2)
    lr_array = 10**(minlr + (maxlr-minlr)*randn[0: num_try])

    minreg = min(reg)
    maxreg = max(reg)
    reg_array = 10**(minreg + (maxreg-minreg)*randn
        [num_try: 2*num_try])
    lr_regs = zip(lr_array, reg_array)
    return lr_regs

def train_random_search(self, lr = [-1, -5], reg = [-1,
    -5], num_try = 10, epoch_more = 1, batch = 64, lr_decay
    = 0.8, mu = 0.9, optimizer = 'Nesterov', regulation
    = 'L2', activation = 'ReLU'):
    self.__methods_check(optimizer, regulation,
        activation)
    self.featuremap_shape()
    lr_regs = self.__gen_lr_reg(lr, reg, num_try)
    for lr_reg in lr_regs:
        try:
            self.init_params()
            self.context = [*lr_reg, batch, lr_decay,
                mu, optimizer, regulation, activation]
            self.__train(epoch_more, *lr_reg, batch,
```

```
                lr_decay, mu, optimizer, regulation,
                activation)
        except KeyboardInterrupt:
            pass

def train_from_checkpoint(self, epoch_more = 10,
    checkpoint_fname = ''):
    self.load_checkpoint(checkpoint_fname)
    [lr, reg, batch, lr_decay, mu, optimizer, regulation,
        activation] = self.context
    lr = np.double(lr)
    reg = np.double(reg)
    batch = np.int(batch)
    lr_decay = np.double(lr_decay)
    mu = np.double(mu)
    self.__train(epoch_more, lr, reg, batch, lr_decay,
        mu, optimizer, regulation, activation)

def test_from_checkpoint(self, checkpoint_fname):
    self.load_checkpoint(checkpoint_fname)
    [lr, reg, batch, lr_decay, mu, optimizer, regulation,
        activation] = self.context
    batch = np.int(batch)
    self.test(batch, activation)

def test(self, batch, activation):
    self.load_test_data()
    accuracys = np.zeros(shape = (self.test_labels.
        shape[0],))
    for i in range(0, self.test_labels.shape[0], batch):
        batch_data = self.test_data[i:i+batch,:]
        label = self.test_labels[i:i+batch]
        accuracys[i:i+batch] = self.predict(batch_data,
            label, activation)

    accuracy = np.mean(accuracys)
    print('the test accuracy: %.5f' % accuracy)
    return accuracy
```

其中，主要函数的功能如下。

- __shuffle_data 用于打乱训练集和验证集的样本。
- __methods_check 用于检查方法的合理性。
- __gen_lr_reg 用于产生随机学习率和正则化系数。
- __train 用于进行训练。每一轮训练时，首先打乱样本，然后取批量样本进行前向、反向和参数更新，最后可视化结果。数据损失、训练和验证批量样本的准确率与第一层权重的更新率进行可视化。每轮训练结束后，保存结果和学习率指数退火。训练结束后，计算测试集的准确率。
- train_random_search 是随机搜索训练方法，其中 self.context = [*lr_reg, batch, lr_decay, mu, optimizer, regulation, activation]保存了训练模型的上下文。
- train_from_checkpoint 从保存的文件中加载模型继续训练。
- test_from_checkpoint 从保存的文件中加载模型来计算测试集准确率。
- test 用于计算测试集的准确率。

9.7 VGG 网络结构

VGG 网络结构类似如下的网络结构：INPUT → [CONV → POOL] → [CONV × 2 → POOL] × 2 → [CONV × 5 → POOL] × 2 → FC × 2 → FC，我们应该如何表示该结构呢？可以采用第 8 章的方法，利用 list 结构存储网络结构，如 struct = ['conv_16_5_2_2'] + ['pool'] +

['conv_32']*2 + ['pool'] + ['conv_64']*3 + ['FC_128']，其中各个参数的意义如下。

- conv 表示卷积层。
- pool 是池化层（步长和窗口大小都是 2）。
- FC 表示全连接层。
- conv_16_5_2_2 具体表示输出 16 个特征图，其卷积核大小为 5，步长为 2，0 填充为 2。
- conv_64 表示输出 64 个特征图，卷积核大小为 3，步长为 1，0 填充为 1。
- FC_128 表示输出神经元有 128 个。

整个结构表示输入层后面接卷积层 conv_16_5_2_2，经过一系列中间层后，FC_128 层后接全连接层得到分值向量。最后会接 softmax 层得到损失。struct = []表示采用线性模型。

```
import numpy as np
import re

class VGGNet(object):
    def __init__(self, struct = []):
        if len(struct) == 0:
            print('you are using linearity model!')
        self.__struct_parse(struct)
        self.__struct = struct
        self.__struct += ['FC', 'softmax']
```

下面的代码对结构 struct 进行解析，采用正则表达式进行字符串匹配，得到每层的超参数（注意，最后全连接层采用 Last_FC 进行标记）：

```
def __struct_parse(self, struct):
    layers = []
    for layer in struct:
        convfull = re.match('^conv_(\d{1,3})_
            (\d{1})_(\d{1})_(\d{1})$', layer)
        convdefault = re.match('^conv_(\d{1,3})$', layer)
        pool = re.match('^pool$', layer)
        fc = re.match('^FC_(\d{1,4})$', layer)
        if convfull:
            layers.append(( int(convfull.group(1)),
                int(convfull.group(2)),
                int(convfull.group(3)), int(convfull.
                    group(4)), 'conv'))
        elif convdefault:
            layers.append(( int(convdefault.group(1)),
                3,1,1,'conv'))
        elif pool:
            layers.append( (layers[-1][0], 'pool') )
        elif fc:
            layers.append( (int(fc.group(1)), 'FC') )
        else:
            raise ValueError('the layer must like conv_16_
                5_2_2 or conv_16 or pool or FC_64')

    layers.append(('', 'Last_FC'))
    self.__layers_params = layers
```

根据 struct 结构和输入图像的属性计算各层特征图的空间尺寸：

```
def featuremap_shape(self):
    maps_shape = []
    in_map_shape = (self.im_height, self.im_width,
        self.im_dims)
    maps_shape.append(in_map_shape)
    for layer in self.__layers_params:
        if layer[-1] == 'Last_FC':
            break
        elif layer[-1] == 'FC':
```

```
                in_map_shape = (1, 1, layer[0])
            elif layer[-1] == 'conv':
                (out_depth, filter_size, stride, padding,
                    not_used) = layer
                out_height = (in_map_shape[0] - filter_size
                    + 2*padding)//stride + 1
                out_width = (in_map_shape[1] - filter_size +
                    2*padding)//stride + 1
                in_map_shape = (out_height, out_width,
                    out_depth)
                if out_height < filter_size or out_width <
                    filter_size:
                    raise ValueError('the cnn struct is not
                        compatible with the image size!\n')
            elif layer[-1] == 'pool':
                filter_size = 2
                stride = 2
                out_height = (in_map_shape[0] - filter_size)
                    //stride + 1
                out_width = (in_map_shape[1] - filter_size)
                    //stride + 1
                in_map_shape = (out_height, out_width, layer[0])
                if out_height < filter_size or out_width <
                    filter_size:
                    raise ValueError('the cnn struct is not
                        compatible with the image size!\n')
            else:
                pass
            maps_shape.append(in_map_shape)
        self.maps_shape = maps_shape

    def init_params(self): # 网络的权重和偏置初始化
        self.__weights = []
        self.__biases = []
        in_depth = self.im_dims
        out_depth = in_depth
        for layer_param, map_shape in zip(self.__layers_params,
            self.maps_shape):
            weight = np.array([])
```

```
            bias = np.array([])
            if layer_param[-1] == 'Last_FC':
                in_depth = out_depth
                out_depth = self.num_class
                (weight, bias) = self.param_init(out_depth,
                    in_depth, map_shape[0]*map_shape[1])
            elif layer_param[-1] == 'FC':
                out_depth = layer_param[0]
                in_depth = map_shape[2]
                (weight, bias) = self.param_init(out_depth,
                    in_depth, map_shape[0]*map_shape[1])
            elif layer_param[-1] == 'conv':
                filter_size = layer_param[1]
                out_depth = layer_param[0]
                (weight, bias) = self.param_init(out_depth,
                    in_depth, filter_size*filter_size)
            elif layer_param[-1] == 'pool':
                pass
            else:
                pass
            in_depth = out_depth
            self.__weights.append(weight)
            self.__biases.append(bias)

        self.__vweights = []
        self.__vbiases = []
        self.__cache_biases = []
        self.__cache_weights = []
        for weight, bias in zip(self.__weights, self.__biases):
            self.__vweights.append(np.zeros_like(weight))
            self.__vbiases.append(np.zeros_like(bias))
            self.__cache_weights.append(np.zeros_like (weight))
            self.__cache_biases.append(np.zeros_like(bias))

    def reg_loss(self, reg=10**(-5), regulation='L2'):
        # 计算正则化损失
        reg_loss = 0
        for weight in self.__weights:
            if weight.size !=  0:
```

```
            reg_loss += self.norm_reg(weight, reg, regulation)
        return reg_loss

    def forward(self, batch_data, labels, reg=10**(-5),
        regulation='L2', activation='ReLU'): # 前向计算
        self.__matric_data = []
        self.__filter_data = []
        self.__matric_data_max_pos = []

        in_maps = batch_data
        for layer_param, weight, bias in zip(self.__layers_params,
            self.__weights, self.__biases):
            matric_data = np.array([])
            filter_data = np.array([])
            matric_data_max_pos = np.array([])
            if layer_param[-1] == 'Last_FC':
                # 最后全连接层不需要激活
                (matric_data, filter_data, out_maps) =
                    self.FC_layer(in_maps, weight, bias,
                    self.num_class, 1, activation)
            elif layer_param[-1] == 'FC':
                (matric_data, filter_data, out_maps) =
                    self.FC_layer(in_maps, weight, bias,
                    layer_param[0], 0, activation)
            elif layer_param[-1] == 'conv':
                (matric_data, filter_data, out_maps) =
                    self.conv_layer(in_maps, weight, bias,
                    layer_param[0:-1], activation)
            elif layer_param[-1] == 'pool':
                (out_maps, matric_data_max_pos) =
                    self.pooling_layer(in_maps)
            else:
                pass
            in_maps = out_maps

            self.__matric_data.append(matric_data)
            self.__filter_data.append(filter_data)
            self.__matric_data_max_pos.append
```

```
                (matric_data_max_pos)

        self.__probs = self.softmax_layer(out_maps)
        data_loss = self.data_loss(self.__probs, labels)
        reg_loss = self.reg_loss(reg, regulation)
        return (data_loss, reg_loss)

    def predict(self, batch_data, labels, activation='ReLU'):
        # 计算批量样本的准确率
        in_maps = batch_data
        for layer_param, weight, bias in zip(self.__layers_
            params, self.__weights, self.__biases):
            if layer_param[-1] == 'Last_FC':
                # 最后全连接层不需要激活
                (matric_data, filter_data, out_maps) =
                    self.FC_layer(in_maps, weight, bias,
                    self.num_class, 1, activation)
            elif layer_param[-1] == 'FC':
                (matric_data, filter_data, out_maps) =
                    self.FC_layer(in_maps, weight, bias,
                    layer_param[0], 0, activation)
            elif layer_param[-1] == 'conv':
                (matric_data, filter_data, out_maps) =
                    self.conv_layer(in_maps, weight, bias,
                    layer_param[0:-1], activation)
            elif layer_param[-1] == 'pool':
                (out_maps, matric_data_max_pos) =
                    self.pooling_layer(in_maps)
            else:
                pass
            in_maps = out_maps
        predicted_class = np.argmax(out_maps, axis = 3)
        accuracy = predicted_class.ravel() == labels
        return np.mean(accuracy)

    def dweight_reg(self, reg = 10**(-5), regulation = 'L2'):
        # 正则化梯度
        for i in range(len(self.__weights)):
```

```
            weight = self.__weights[i]
            if weight.size ! =  0:
                self.__dweights[-1-i] += self.dnorm_reg
                     (weight, reg, regulation)

    def backpropagation(self, labels, reg = 10**(-5),
        regulation = 'L2', activation = 'ReLU'): # 反向传播
        dscores = self.evaluate_dscores(self.__probs, labels)
        dout_maps = dscores
        self.__dweights = []
        self.__dbiases = []
        for (layer_param, maps_shape, weight,
            matric_data, filter_data, matric_data_max_pos)
                in zip(reversed(self.__layers_params),
            reversed(self.maps_shape), reversed(self.__weights),
            reversed(self.__matric_data), reversed(self.
                __filter_data), reversed(self.__matric_data_
                max_pos) ):
            if layer_param[-1] == 'Last_FC':
                (dweight, dbias, din_maps) = self.dFC_layer
                    (dout_maps, matric_data, filter_data,
                        weight, maps_shape, 1, activation)
            elif layer_param[-1] == 'FC':
                (dweight, dbias, din_maps) = self.dFC_layer
                    (dout_maps, matric_data, filter_data,
                        weight, maps_shape, 0, activation)
            elif layer_param[-1] == 'conv':
                (dweight, dbias, din_maps) =
                    self.dconv_layer(dout_maps, matric_data,
                        filter_data, weight, maps_shape,
                        layer_param[1:-1], activation)
            elif layer_param[-1] == 'pool':
                dweight = np.array([])
                dbias = np.array([])
                din_maps = self.dpooling_layer(dout_maps,
                    matric_data_max_pos, maps_shape)
            else:
                pass
```

```
            dout_maps = din_maps
            self.__dweights.append(dweight)
            self.__dbiases.append(dbias)
            self.__dbatch_data = din_maps # 输入图像的梯度
            self.dweight_reg(reg, regulation)

    def params_update(self, lr=10**(-4), t=1, mu=0.9,
        optimizer='Nesterov'): # 参数更新
        self.update_ratio = []
        if optimizer == 'adam':
            for i in range(len(self.__weights)):
                weight = self.__weights[i]
                bias = self.__biases[i]
                dweight = self.__dweights[-1-i]
                dbias = self.__dbiases[-1-i]
                v_weight = self.__vweights[i]
                v_bias = self.__vbiases[i]
                cache_weight = self.__cache_weights[i]
                cache_bias = self.__cache_biases[i]
                if weight.size != 0:
                    (update_ratio_w, weight, v_weight,
                        cache_weight) = self.adam(lr,
                        weight, v_weight, cache_weight,
                        dweight, t, mu)
                    (update_ratio_b, bias, v_bias,
                        cache_bias) = self.adam(lr, bias,
                        v_bias, cache_bias, dbias, t, mu)
                    self.__weights[i] = weight
                    self.__biases[i] = bias
                    self.__vweights[i] = v_weight
                    self.__vbiases[i] = v_bias
                    self.__cache_weights[i] = cache_weight
                    self.__cache_biases[i] = cache_bias
                    self.update_ratio.append((update_
                        ratio_w,update_ratio_b))

        if optimizer == 'Nesterov':
            for i in range(len(self.__weights)):
                weight = self.__weights[i]
```

```
                bias = self.__biases[i]
                dweight = self.__dweights[-1-i]
                dbias = self.__dbiases[-1-i]
                v_weight = self.__vweights[i]
                v_bias = self.__vbiases[i]
                if weight.size != 0:
                    (update_ratio_w, weight, v_weight) =
                        self.nesterov_momentumGD(lr,
                        weight, v_weight, dweight, mu)
                    (update_ratio_b, bias, v_bias) =
                        self.nesterov_momentumGD(lr, bias,
                        v_bias, dbias, mu)
                    self.__weights[i] = weight
                    self.__biases[i] = bias
                    self.__vweights[i] = v_weight
                    self.__vbiases[i] = v_bias
                    self.update_ratio.append((update_
                        ratio_w,update_ratio_b))

    def save_checkpoint(self, fname): # 保存模型所有数据
        with open(fname, 'wb') as f:
            np.save(f, np.array([3,1,4,1,5,9,2,8,8])) # 魔术数
            np.save(f, np.array( self.__struct) ) # 网络结构
            np.save(f, np.array([self.num_class, self.im_dims,
                self.im_height, self.im_width]) )
                # 输入图像属性和类别数
            np.save(f, np.array(self.__layers_params))
                # 每层超参数
            np.save(f, np.array(self.maps_shape)) # 特征图空间尺寸
            np.save(f, np.array(self.context)) # 训练环境上下文
            for array in self.__weights: # 网络参数
                np.save(f, array)
            for array in self.__biases:
                np.save(f, array)
            for array in self.__vweights:
                np.save(f, array)
            for array in self.__vbiases:
                np.save(f, array)
```

```
            for array in self.__cache_weights:
                np.save(f, array)
            for array in self.__cache_biases:
                np.save(f, array)

    def load_checkpoint(self, fname): # 载入模型所有数据
        with open(fname, 'rb') as f:
            magic_number = np.load(f)
            if not all(magic_number == np.array([3,1,4,1,5,
                9,2,8,8])):  # 魔术数
                raise ValueError('the file format is wrong!\n')
                self.__struct = np.load(f)
                print('\n\nthe net struct is: \n', self.__struct)
                im_property = np.load(f)
                self.num_class, self.im_dims, self.im_height,
                    self.im_width = im_property
                self.__layers_params = np.load(f)
                self.maps_shape = np.load(f)
                self.context = np.load(f)
                self.__weights = []
                self.__biases = []
                for i in range(len(self.__layers_params)):
                    array = np.load(f)
                    self.__weights.append(array)
                for i in range(len(self.__layers_params)):
                    array = np.load(f)
                    self.__biases.append(array)
                self.__vweights = []
                self.__vbiases = []
                for i in range(len(self.__layers_params)):
                    array = np.load(f)
                    self.__vweights.append(array)
                for i in range(len(self.__layers_params)):
                    array = np.load(f)
                    self.__vbiases.append(array)
                self.__cache_weights = []
                self.__cache_biases = []
                for i in range(len(self.__layers_params)):
                    array = np.load(f)
```

```
                self.__cache_weights.append(array)
            for i in range(len(self.__layers_params)):
                array = np.load(f)
                self.__cache_biases.append(array)
            print('the struct hyper parameters:\n',
                self.__layers_params)
```

9.8　MNIST 数据集

在 http://yann.lecun.com/exdb/mnist/上下载 MNIST 数据集，并将其保存在 MNIST\ 目录下。预处理该数据集时，只是简单地归一化到 [0, 1]。相关代码如下：

```
import numpy as np
import gzip, struct

class MNISTInterface(object):
    '''
    load the mnist dataset
    and shuffle split the train set into train and validation
    set the ratio of train and validation may be 7:3
    '''
    # 载入训练集，并随机分为训练集和验证集
    def load_train_data(self, num_ratio):
        (imgs, labels) = MNISTInterface.get_mnist_train()
        # 数据预处理
        imgs = imgs/255  # 归一化到[0, 1]
        self.num_samples = labels.size
        if isinstance(num_ratio, int):
            self.num_train_samples = num_ratio
        else:
            self.num_train_samples =
                int(self.num_samples*num_ratio)
        self.num_val_samples = self.num_samples -
```

```
                self.num_train_samples
            shuffle_no = list(range(self.num_samples))
            np.random.shuffle(shuffle_no)
            imgs = imgs[shuffle_no]
            labels = labels[shuffle_no]
            self.train_data = imgs[0:self.num_train_samples]
            self.train_labels = labels[0:self.num_train_samples]
            self.val_data = imgs[self.num_train_samples::]
            self.val_labels = labels[self.num_train_samples::]
            self.__set_data_pro()
    # 载入测试集
    def load_test_data(self):
        (imgs, labels) = MNISTInterface.get_mnist_test()
        # 数据预处理
        imgs = imgs/255  # 归一化到[0, 1]
        self.test_data = imgs
        self.test_labels = labels
        self.__set_data_pro()
    # 设置数据集超参数
    def __set_data_pro(self, num_class=10, im_height=28,
        im_width=28, im_dims=1):
        self.num_class = num_class
        self.im_height = im_height
        self.im_width = im_width
        self.im_dims = im_dims
    # 读图像和标签数据
    @staticmethod
    def __read(image, label):
        mnist_dir = 'MNIST/' # 存储数据集的目录
        with gzip.open(mnist_dir + label) as flbl:
            magic, num = struct.unpack(">II", flbl.read(8))
            label = np.fromstring(flbl.read(), dtype = np.uint8)
        with gzip.open(mnist_dir + image, 'rb') as fimg:
            magic, num, rows, cols = struct.unpack(">IIII",
                fimg.read(16))
            image = np.fromstring(fimg.read(), dtype = np.uint8).
                reshape(len(label), rows, cols)
        return (image, label)
```

```
@staticmethod
def get_mnist_train():
    train_img, train_label = MNISTInterface.__read
       ('train-images-idx3-ubyte.gz',
        'train-labels-idx1-ubyte.gz')
    train_img = train_img.reshape((*train_img.shape,1))
        # 转变为 4D 特征图
    return (train_img, train_label)

@staticmethod
def get_mnist_test():
    test_img, test_label = MNISTInterface.__read
        ('t10k-images-idx3-ubyte.gz',
        't10k-labels-idx1-ubyte.gz')
    test_img = test_img.reshape((*test_img.shape,1))
    return (test_img, test_label)
```

9.9　梯度检测

梯度检测代码如下：

```
import numpy as np

from vgg_net import VGGNet
from cnn_block_interface import CnnBlockInterface
from regulation_interface import RegulationInterface

class VGGTest(VGGNet, CnnBlockInterface, RegulationInterface):
    def set_data_pro(self, num_class=4, im_height=32,
        im_width=32, im_dims=3):
        self.num_class = num_class
        self.im_height = im_height
        self.im_width = im_width
        self.im_dims = im_dims

    def gen_random_data(self):
```

```
        self.num_samples = self.num_class*20 # 每类 20 个样本
        self.data = np.random.randn(self.num_samples,
            self.im_height, self.im_width, self.im_dims)
        self.labels = np.random.randint(self.num_class,
            size = self.num_samples)

    def check_gradient(self, check_weight_or_bias=1,
        step=10**(-5), reg=10**(-1), regulation='L1',
        activation='ELU'):
        # check_weight_or_bias 等于 1 时，检测权重；等于 0 时，
          检测偏置
        self.set_data_pro()
        self.gen_random_data()
        self.featuremap_shape()
        self.init_params()
        for layer in range(len(self.maps_shape)): # 每层都检测
            if check_weight_or_bias:
                weight = self._VGGNet__weights[layer]
                    # 注意如何引用私有成员
                if weight.size == 0:
                    continue
                else:
                    row = np.random.randint(weight.shape[0])
                    col = np.random.randint(weight.shape[1])
                    param = weight[row][col]
            else:
                bias = self._VGGNet__biases[layer]
                if bias.size == 0:
                    continue
                else:
                    row = np.random.randint(bias.shape[1])
                    param = bias[0][row]

        (data_loss, reg_loss) = self.forward(self.data,
            self.labels, reg, regulation, activation)
        self.backpropagation(self.labels, reg, regulation,
            activation)
        if check_weight_or_bias:
            danalytic = self._VGGNet__dweights[-1-layer]
```

```
                [row][col]
        else:
            danalytic = self._VGGNet__dbiases[-1-layer]
                [0][row]

        if check_weight_or_bias:
            self._VGGNet__weights[layer][row][col] =
                param - step
        else:
            self._VGGNet__biases[layer][0][row] = param - step
        (data_loss1, reg_loss) = self.forward(self.data,
            self.labels, reg, regulation, activation)
        loss1 = data_loss1 + reg_loss

        if check_weight_or_bias:
            self._VGGNet__weights[layer][row][col] =
               param + step
        else:
            self._VGGNet__biases[layer][0][row] = param
                + step
        (data_loss2, reg_loss) = self.forward(self.data,
            self.labels, reg, regulation, activation)
        loss2 = data_loss2 + reg_loss
        dnumeric = (loss2 - loss1)/(2*step)

        print(layer, data_loss1, data_loss2)
        error_relative = np.abs(danalytic - dnumeric)/
            np.maximum(danalytic, dnumeric)
        print(danalytic, dnumeric, error_relative)

if __name__ == '__main__':
    # 网络结构
    struct = ['conv_32_5_1_0'] + ['pool'] + ['conv_64'] +
        ['pool']+ ['conv_128']*2 + ['pool'] + ['conv_256'] +
        ['FC_100']
    vgg = VGGTest(struct) # 创建网络实例
    vgg.check_gradient(check_weight_or_bias=1,
        step=10**(-5), reg=10**(-5), regulation='L2',
        activation='ELU')
```

采用上面的参数设置，梯度的相对误差在 10**(-9)左右时，说明梯度计算正确。读者可以采用不同的参数组合进行测试。

9.10 MNIST 数据集的训练结果

MNIST 数据集的训练：

```
from vgg_net import VGGNet
from cnn_block_interface import CnnBlockInterface
from cnn_train_interface import CnnTrainInterface
from optimizer_interface import OptimizerInterface
from regulation_interface import RegulationInterface
from MNIST_interface import MNISTInterface

class VGGTest(MNISTInterface, VGGNet, CnnBlockInterface,
CnnTrainInterface, OptimizerInterface, RegulationInterface):
    pass # 无须任何代码

if __name__ == '__main__':
#    struct = [] # 线性模型
#    struct = ['FC_64'] # 只有一个隐含层的神经网络

    struct = ['conv_8'] + ['pool'] + ['conv_12']*3 + ['pool'] +
        ['conv_36']*3  + ['pool'] + ['FC_64']
    vgg = VGGTest(struct)
    num_samples = 0.7
    vgg.load_train_data(num_samples)
    train = 1
    scratch = 1
    if train:
        if scratch:
            vgg.train_random_search(lr=[-1.0, -5.0],
                reg=[-3, -5], num_try=10, epoch_more=2,
                batch=64, lr_decay=1, mu=0.9,
                optimizer='adam', regulation='L2',
```

```
            activation='ELU') # 超参数随机搜索
    else:
        vgg.train_from_checkpoint(epoch_more=2,
            checkpoint_fname='checkpoint_
            (loss_-1.07)_(epoch_2) __[(lr reg)_(-3.0
            -3.0)]_ adam L2 ELU.npy') # 从保存点继续训练
else:
# 从保存点计算测试集准确率
    vgg.test_from_checkpoint('checkpoint_
        (loss_-1.0)_(epoch_4)__[(lr reg)_(-4.0 -4.0)]_
        adam L2 ELU.npy')
```

下面给出一些有趣的结果。采用上面的网络结构，训练两轮，测试集的准确率在 97.5% 左右，训练过程的性能曲线如图 9.1 所示，准确率上升很快，损失值波动很大，过拟合不明显，参数更新率在 `10**(-3.5)`左右。这表明学习率大了，减小学习率，需要训练 8 轮左右，最终准确率可达 98.7% 左右。

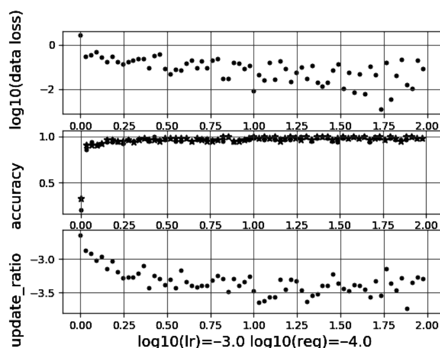


图 9.1　训练过程（另见彩插）

采用 `struct = []`线性模型时，训练两轮，测试集的准确率在 92% 左右，训练过程的性能曲线如图 9.2 所示，准确率上升很快，损失值波动很大，过拟合不明显，参数更新率在 `10**(-3)`左右。

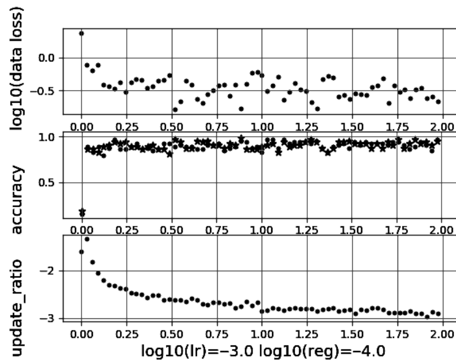


图 9.2　线性模型的训练过程（另见彩插）

采用 `struct = ['FC_64']`，训练两轮，测试集的准确率在 95.5% 左右，训练过程的性能曲线如图 9.3 所示，准确率上升很快，损失值波动很大，过拟合不明显，参数更新率在 `10**(-2.5)`左右。

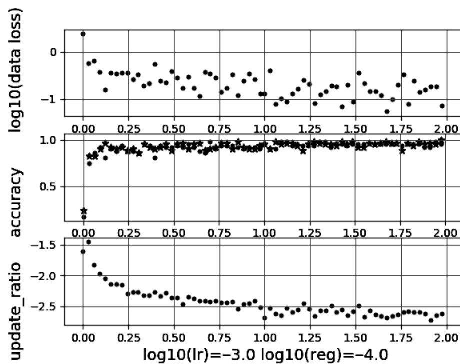


图 9.3　一个隐含层（64 个神经元）神经网络模型的训练过程（另见彩插）

上面的结果都是训练集样本数比例为 0.7 得到的，现在采用不同的训练样本数进行训练，会得到一些有趣结果。此时模型都为 `struct = ['conv_8'] + ['pool'] + ['conv_12']*3 + ['pool'] +`

['conv_36']*3+ ['pool'] + ['FC_64']，当训练样本数为 40 时，测试集的准确率达 63.6%，400 个达到 86.7%，3000 个达到 93.0%，6000 个达到 96.3%。

9.11　程序使用建议

如果读者只采用类似于 VGG 的网络进行训练，那么使用本程序只需改变网络结构，如设置 struct = ['conv_64'] + ['pool'] + ['conv_128']*2 + ['pool']+ ['conv_256']*3+ ['pool'] + ['conv_512']*3 + ['FC_1024'] + ['FC_1024']和使用自己的数据集及数据预处理，然后直接调用超参数随机搜索函数进行训练，保存最好模型的参数即可。如果对卷积网络进行改进，只需要修改 9.7 节，其他模块基本无须修改。需要注意，Python 类与 C++的静态类（其所有属性和方法在定义时必须全部声明，结构很清晰）不同，它是动态类，其成员属性和方法是运行时绑定的，在定义时无须绑定，这个特性使设计类十分灵活，但也很容易出现逻辑错误，造成理解困难。举个例子，在 CnnTrainInterface 类中，方法 __train 调用了方法 forward，但是该类中并没有定义 forward，也没有继承 forward，程序仍能正常运行，使人如坠雾里。其实，forward 是调用了 VGGNet 类的实现，因此最终调用 forward 的并不是 CnnTrainInterface 类，而是 VGGTest。VGGTest 类继承了 VGGNet 类，可以调用 forward，这就是运行时绑定机制。因为有动态绑定机制，所以类的设计可以十分独立，不需要复杂的继承关系。

第 10 章

卷积网络结构的发展

10.1 全局平均池化层

第 4 章介绍了卷积网络的基本结构：首先堆叠一个或多个卷积层，然后接一个池化层，重复此模式，最后接多个全连接层。随着网络加深，特征图深度变大，空间尺寸变小，最后一个特征图的空间尺寸一般为 7×7。这种模式存在一个缺点，那就是最后一个卷积层变换为全连接层时所需要的参数特别多。以 VGG 网络为例，最后一个特征图的空间尺寸是 7×7，深度是 512，后面全连接层有 4096 个神经元，这个变换需要的权重数量为 $7 \times 7 \times 512 \times 4096 = 102\,760\,448$，大得惊人，且占到总参数 138M 的 74%。VGG 网络后面还有两个全连接层，权重数量分别为 $4096 \times 4096 = 16\,777\,216$ 和 $4096 \times 1000 = 4\,096\,000$，这 3 个全连接层的权重数量占到总参数的 90%，参数巨大，不仅消耗大量的内存和计算资源，而且容易引起过拟合，导致泛化性能变差。分析后面全连接层的作用，是把 $7 \times 7 \times 512$ 特征图变换为 1000 维的分值向

量。为了达到这个目的，我们可以分两步：先把 $7\times7\times512$ 特征图变换为 $1\times1\times512$ 的特征图，然后接 1000 个神经元的全连接层得到分值向量。$7\times7\times512$ 特征图变换为 $1\times1\times512$ 的特征图，可以不需要任何参数，只需对每个 7×7 特征图进行处理，得到一个输出值，犹如池化层。池化是 2×2 的空间得到一个输出值，一般采用最大值或平均值。这里采用 7×7 特征图的平均值作为输出值，称为全局平均池化层。这样，把 $7\times7\times512$ 特征图变换为 1000 维的分值向量，需要的权重数量仅为 $512\times1000=512\ 000\approx0.5\text{M}$！全局平均池化层的学习容量显然低于全连接层，虽然从表面看会降低网络性能，但实际上由于参数大幅减小，不容易过拟合，全局平均池化层对性能几乎没有影响，甚至有时效果更好。全局平均池化层还有一个额外的好处，即不论网络输入的图像尺寸是多少，最后一个特征图均可表示为 $h\times w\times d$，对每个 $h\times w$ 取平均值，所以输出总是 d 维向量，再接全连接层，得到分值向量。

全局平均池化层有两种实现方式，一种如前面例子所示，全局平均池化层后接全连接层得到分值向量；另一种是全局平均池化层的输出直接作为分值向量，此时只需最后一层特征图的深度等于类别数。第一种方式的优点是网络结构比较灵活，全连接层前提取的特征更具有普遍性，迁移能力强，所以实践中多采用第一方式。

全局平均池化层的代码如下：

```
def global_average_pooling_layer(in_data):
    '''
    in_data.shape = [batch,in_height,in_width,in_depth]
    out_data.shape = [batch,out_height = 1,out_width = 1,
```

```
        out_depth = in_depth]
    '''
    (batch,in_height,in_width,in_depth) = in_data.shape
    out_height = 1
    out_width =  1
    out_depth = in_depth
    out_data = np.zeros((batch, out_height, out_width,
        out_depth))

    for i_batch in range(batch):
        for i_map in range(in_depth):
            out_data[i_batch,:,:,i_map] = np.mean(
                in_data[i_batch,:,:,i_map])
    return out_data

def dglobal_average_pooling_layer(dout_data, maps_shape):
    '''
    dout_data.shape = [batch,out_height = 1,out_width = 1,
        out_depth = in_depth]
    '''
    (in_height,in_width,in_depth) = maps_shape
    batch = dout_data.shape[0]
    din_data = np.zeros((batch,in_height,in_width,
        in_depth))

    in_size = in_height*in_width
    for i_batch in range(batch):
        for i_map in range(in_depth):
            din_data[i_batch,:,:,i_map] = dout_data
                [i_batch,:,:,i_map]/in_size
    return din_data
```

10.2 去掉池化层

第 4 章介绍了池化层的主要目的是减小特征图的空间尺寸，其实

采用步长为 2 的卷积层同样可以减小特征图的空间尺寸，所以池化层就没有存在的必要了。有两种取代池化层的方式：第一种是把池化层的前一个卷积层的步长变为 2；第二种是额外增加一个卷积层来取代池化层。举例如下，假设原始网络为 `struct = ['conv_8'] + ['pool'] + ['FC_64']`，则第一种方式的网络变为 `struct = ['conv_8_3_2_1'] + ['FC_64']`，第二种方式的网络为 `struct = ['conv_8'] + ['conv_8_3_2_1'] + ['FC_64']`。两种方式各有优缺点：第一种方式的优点是不需增加参数数量，前一个卷积层由于步长为 2，计算量减小，缺点是由于步长为 2，损失了部分信息，会使网络性能稍微变差一些；第二种方式的优点是没有损失任何信息，不会使网络性能变差，缺点是额外增加了一个卷积层，因此参数数量与计算量增加。

10.3　网络向更深更宽发展面临的困难

在深度学习时代，网络规模越来越大，这表现在网络的深度加深，宽度加大。但如果只是简单地加深或加宽，不一定能带来性能的提升。采用基本结构，每个卷积层变换需要的权重数量为 in_depth × F × F × out_depth，网络变宽，则超参数 in_depth 和 out_depth 都会变大，导致参数数量变大，计算量增大且容易导致过拟合。网络变深时，如 6.3 节的误差反向传播算法所示，损失对权重的梯度包含激活函数梯度的连乘积，网络越深，该乘积为 0 的概率越高，导致大部分梯度为 0，权重不能得到有效更新，难以优化深层网络。

10.4 ResNet 向更深发展的代表网络

ResNet 结构通过学习残差，有效克服了深层网络的优化难题。为了对比，这里先把 6.3 节的内容重复一遍。令深度神经网络的每层神经元的数量为 1，忽略偏置。神经网络用于回归任务，则数学模型为：

$$
\begin{gathered}
a_1 = xw_1, h_1 = f(a_1) \\
\vdots \\
a_i = h_{i-1}w_i, h_i = f(a_i) \\
\vdots \\
a_n = h_{n-1}w_n, h_n = a_n \\
\text{loss} = \frac{1}{2}(a_n - y_0)^2
\end{gathered}
\tag{10.1}
$$

注意，上式中令 h_0 等于 x，$f'(a_n)$ 等于 1。其中 x 是输入，y_0 是 x 对应的标签，即需拟合的真实值，h_n 是网络的预测值，$f(x)$是非线性激活函数。对每层求偏导数，然后进行连乘，得到损失 loss 对任一层参数 w_i 的梯度：

$$
\frac{\partial \text{loss}}{\partial w_i} = (h_n - y_0)\prod_{j=i+1}^{n} w_j \prod_{j=i}^{n} f'(a_j)h_{i-1} \tag{10.2}
$$

上面的网络中，每一层的模型都是 $h_i = f(h_{i-1}w_i)$，称为常规模型。

如果改为下面的模型：

$$
\begin{gathered}
a_1 = f(x)w_1, h_1 = a_1 + x \\
\vdots \\
a_i = f(h_{i-1})w_i, h_i = a_i + h_{i-1} \\
\vdots \\
a_n = f(h_{n-1})w_n, h_n = a_n + h_{n-1} \\
\text{loss} = \frac{1}{2}(a_n - y_0)^2
\end{gathered} \tag{10.3}
$$

则网络每一层的模型都是 $h_i = f(h_{i-1})w_i + h_{i-1}$，称为残差模型，这是因为 $f(h_{i-1})w_i = h_i - h_{i-1}$ 只需学习输入和输出之间的差。

进行简单代入，可以得到任一层的输出 h_i：

$$
h_i = \sum_{i=0}^{i-1} f(h_i)w_{i+1} + h_0 \tag{10.4}
$$

其中令 h_0 等于 x，可见输入 h_0 与任一层输出直接相连，这种连接方式称为直连（shortcut），直连能极大地降低优化难度。而常规模型的输出为：

$$
h_i = f(w_i f(w_{i-1} \cdots f(w_1 h_0))) \tag{10.5}
$$

输入 h_0 不与任一层输出直接相连，而是嵌套相连。

同样采用链式法则，对每层求偏导数，然后进行连乘，得到损失 loss 对任意一层参数 w_i 的梯度：

$$
\frac{\partial \text{loss}}{\partial w_i} = (h_n - y_0)\prod_{j=i+1}^{n}(1 + w_j f'(h_{j-1}))f(h_{i-1}) \tag{10.6}
$$

当激活函数采用 ReLU 时，如果梯度为 0，则单个乘积项为 1；如果激活函数的梯度为 1，则单个乘积项为 $(1+w_j)$。由于权重参数较小，该

项在 1 附近。这样总的乘积 $\prod_{j=i+1}^{n}(1+w_j f'(h_{j-1}))$ 在 1 附近，避免了梯度爆炸（绝对值很大）或消失（绝对值趋向 0），而且和网络深度 n 无关。所以这种模型能优化很深的网络，例如 ResNet 最深有 1001 层，而在常规模型中，网络深度达到 50 层时就很难优化了。

对于残差模型 $h_i=f(h_{i-1})w_i+h_{i-1}$，一定注意激活和权重相乘的顺序，先激活再权重相乘。如果反过来，则模型为 $h_i=f(h_{i-1}w_i)+h_{i-1}$。当激活函数选为最常用的 ReLU 时，因为其输出值大于等于 0，所以 $h_i \geqslant h_{i-1}$，其表达能力就受限了，导致网络学习容量低。

上面这个例子中，隐含层只有一个神经元，当隐含层是向量时，只需把权重看成矩阵即可：

$$h_i = f(h_{i-1})\boldsymbol{W}_i + h_{i-1} \tag{10.7}$$

此时输入层 h_{i-1} 和输出层 h_i 的维度必须相等。如果不相等，则需对输入层 h_{i-1} 进行投影，即用变换矩阵 $\boldsymbol{W}_s$ 把输入层 h_{i-1} 的维度变为与输出层 h_i 的维度相等：

$$h_i = f(h_{i-1})\boldsymbol{W}_i + h_{i-1}\boldsymbol{W}_s \tag{10.8}$$

如果需进行批量标准化，则残差模型为：

$$h_i = f(\mathrm{BN}(h_{i-1}))\boldsymbol{W}_i + h_{i-1} \tag{10.9}$$

一定要注意顺序：先批量标准化，后非线性激活，最后权重相乘。如果是卷积网络，则只需把权重相乘变成卷积运算即可。

在实际应用中，注意需先对输入 x 进行权重相乘后，才能采用残差模块。

10.5　GoogLeNet 向更宽发展的代表网络

GoogLeNet 通过使用 1×1 的卷积核、多尺度卷积核和特征图深度方向串连等技术，达到网络宽度不变的同时，参数数量和计算量相比基本结构也减小了。其基本结构如图 10.1 所示，称为 Inception 模块。

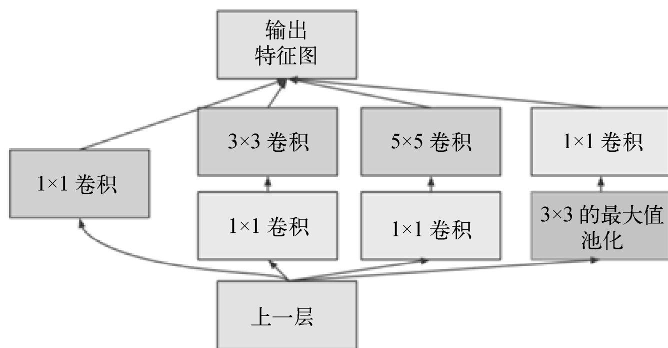


图 10.1　Inception 模块

下面举例说明 Inception 模块的工作方式。假设输入的特征图为 $28 \times 28 \times 192$，输出特征图为 $28 \times 28 \times 256$。如果采用基本结构，卷积核尺寸为 3×3，则需要的权重数量为 $192 \times 3 \times 3 \times 256 = 442\ 368$。Inception 模块通过 3 路卷积和 1 路池化得到输出特征图，具体为：第 1 路是 1×1 卷积，输出 64 个特征图，所需权重数量为 $192 \times 1 \times 1 \times 64 = 12\ 288$；第 2 路是先进行 1×1 卷积以进行降维，得到 96 个特征图，然后进行 3×3 卷积，输出 128 个特征图，该路所需的权重数量为 $192 \times 1 \times 1 \times 96 + 96 \times 3 \times 3 \times 128 = 129\ 024$；第 3 路和第 2 路类似，只是采用了 5×5 的卷积核，具体为：1×1 卷积得到 16 个特征图，然后

进行 5×5 卷积，输出 32 个特征图，所需的权重数量为 192×1×1×16+16×5×5×32 = 15 872；最后 1 路是先进行 3×3 的最大值池化（注意步长为 1，因为不需要降低空间尺寸），然后 1×1 卷积得到 32 个特征图，所需的权重数量为 192×1×1×32 = 6144；最后输出的特征图是这 4 路输出特征图的串连，所以特征图数量是这 4 路输出之和：64+128+32+32 = 256，总的权重数量为 12 288+129 024+15 872+ 6144 = 163 328，是基本结构权重数量 442 368 的 37%，有效减小了权重数量。注意，每个卷积都紧跟非线性激活，包括 4 个 1×1 的卷积。

每个卷积层变换需要的权重数量为 `in_depth × F × F × out_depth`，在 `in_depth` 和 `out_depth` 不变的情况下，1×1 卷积核只是 3×3 卷积核权重数量的 1/9，这是 Inception 模块能减小参数数量的基础。通过 1×1 卷积进行深度降维，减小 3×3 和 5×5 卷积层输入 `in_depth` 的大小，从而减小参数数量，这也是广泛使用 1×1 卷积的原因。同时使用 3 种卷积核尺寸：1×1、3×3 和 5×5，得到特征图，这是为了提取输入特征图的多尺度特征。一般情况下，3×3 输出的特征图数量最多，1×1 其次，5×5 最少，如例子的 128、64 和 32。此外，我们还可以进一步减小权重数量，如把 5×5 的卷积分成 2 个 3×3 的卷积，2×3×3/5×5 = 18/25；更进一步地说，把 3×3 的卷积分成 1×3 和 3×1 两个卷积，(1×3+3×1)/3×3 = 2/3。

一般情况下，Inception 模块的输入特征图的空间尺寸在 35×35 以下，如果特征图的空间尺寸较大，仍采用基本结构。

10.6　轻量网络

轻量网络是指网络的参数数量和计算量少，能在移动系统（如手机）中使用，同时网络性能尽可能高。目前，轻量网络都去掉了池化层，采用平均池化层代替全连接层。对卷积网络的核心卷积层也进行了改进，使其参数和计算量减少。卷积层操作是采用卷积核对特征图的局部数据进行卷积，局部数据是指空间维度数据，但深度维度是全部数据，这个特性在第 4 章中反复强调过。轻量网络采用分离卷积来减小参数数量和计算量，分离卷积是对空间维度和深度维度进行解耦卷积，即先采用 1×1 的卷积核进行深度维度卷积，这和常规卷积一样，对深度维度采用全部数据，但由于卷积核的尺寸是 1×1，所以并没有对空间维度进行卷积。然后采用 3×3 的卷积核进行空间维度卷积，不同于常规卷积，深度维度不采用全部数据，而是逐特征图进行卷积，即每个特征图只进行一个 3×3 的卷积，分别输出一个特征图。对比常规卷积和分离卷积的参数数量和计算量，假设输入和输出特征图都是 $h\times w\times d$，常规卷积时卷积核尺寸是 3×3，则权重数量为 $3\times3\times d\times d=9d^2$，计算量为 $h\times w\times3\times3\times d\times d=9d^2hw$。分离卷积时，深度维度卷积采用 1×1 的常规卷积，则权重数量为 $1\times1\times d\times d=d^2$，计算量为 $h\times w\times1\times1\times d\times d=d^2hw$；空间维度卷积采用逐特征图卷积，卷积核尺寸为 3×3，权重数量为 $3\times3\times d=9d$，计算量为 $h\times w\times3\times3\times d=9dhw$。参数比例为 $(d^2+9d)/9d^2=1/9+1/d$，计算量比例为 $(d^2hw+9dhw)/9d^2hw=1/9+1/d$。由于 d 一般远大于 9，所以分离卷积的参数数量和计算量都是常规卷积的 1/9 到 1/8，分离卷积的主要参数和计算量都来自 1×1 深度维度卷积，3×3 空间维度卷积可忽略不计。

下面以 Google 公司 2018 年提出的 MobileNetV2 网络为例介绍分离卷积的应用，如图 10.2 所示。其中，输入特征图是 $h \times w \times d$。首先，对输入特征图进行步长为 1 的 1×1 深度维度的卷积，得到特征图 $h \times w \times (6d)$。注意，深度维度增大为原来的 6 倍，以提取更丰富的特征。然后，进行非线性激活。接着进行逐特征图的空间卷积，卷积核的尺寸是 3×3，步长为 s。当 $s = 1$ 时，空间尺寸不变；当 $s = 2$ 时，空间尺寸减半，得到特征图 $(h/s) \times (w/s) \times (6d)$，然后进行非线性激活。最后，再一次进行步长为 1 的 1×1 深度维度卷积，得到特征图 $(h/s) \times (w/s) \times d'$，注意此卷积不需要接非线性激活。如果步长 $s = 1$，$d' = d$，则采用 ResNet 结构（即 shortcut 直连），把输入特征图加到最后一个 1×1 卷积层的输出中，作为模块的最终输出。该模块称为逆残差模块（inverted residual block），因为中间特征图的深度维度 $6d$ 大于输入特征图的深度维度 d，所以称为逆（常规残差模块的中间特征图的深度维度更小）。非线性激活采用 $\text{ReLU6} = \min(6, \max(0, x))$，即限制最大输出值小于 6，这是因为轻量网络一般采用单精度浮点数进行计算，当数值大于 6 时，表示精度不高，会造成计算误差。逆残差模块包含 3 个卷积层，每个卷积输出可以采用批量归一化处理，以加快训练速度，提高网络性能。

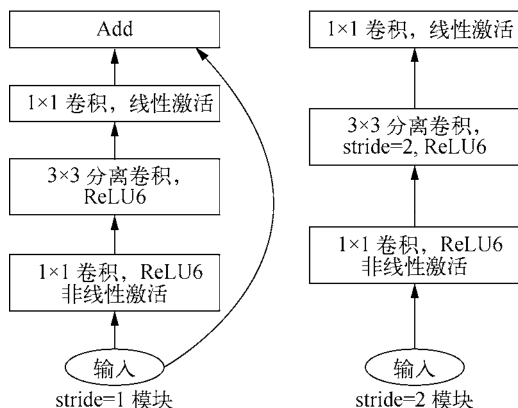


图 10.2　MobileNetV2 网络分离卷积模块——逆残差模块

由于 1 × 1 深度维度卷积的广泛使用和轻量网络的重要性，下面将给出这些算法的 Python 代码实现。请读者注意，这里没有实现批量归一化。

10.6.1　1 × 1 深度维度卷积代码实现

1 × 1 深度维度卷积是常规卷积的特例，所以可以直接采用第 4 章的代码。由于卷积核是 1 × 1，可以不进行特征图矩阵化，而是直接获得 1 × 1 局部窗口的数据，乘以权重矩阵（矩阵乘法），得到输出特征图的 1 × 1 局部窗口数据。这种实现的计算效率远高于常规实现，而且内存消耗很少。其代码和第 4 章很类似，这里直接给出代码：

```
def conv11_layer(in_data, weights, biases, out_depth,
    activation='ReLU'):
    '''
    in_data.shape = [batch,in_height,in_width,in_depth]
    weights.shape = [in_depth, out_depth]
    biases.shape = [1, out_depth]
```

```
    out_data.shape = [batch, in_height, in_width, out_depth]
    '''
    (batch, in_height, in_width, in_depth) = in_data.shape
    out_data = np.zeros((batch, in_height, in_width,
        out_depth))
    for i_batch in range(batch):
        for i_height in range(in_height):
            for i_width in range(in_width):
                out_data[i_batch, i_height, i_width, :]
                    = np.dot(in_data[i_batch, i_height,
                    i_width, :], weights)

    out_data += biases
    # 此处可以加入批量归一化处理代码
    if activation != 'None':
        out_data = CnnBlockInterface.activation
            (out_data, activation)
    return out_data
```

梯度反向传播过程和正向流程相反，只需对非线性激活和矩阵乘法进行梯度计算，代码很简单：

```
def dconv11_layer(dout_data, out_data, in_data, weights,
    activation='ReLU'):
    '''
    inputs: dout_data, in_data
    outputs: (dweights, dbiases, din_data)
    '''
    if activation != 'None':
        dout_data = CnnBlockInterface.dactivation
            (dout_data, out_data, activation)
    dbiases = np.sum(dout_data.reshape(-1,
        dout_data.shape[-1]), axis=0, keepdims=True)
    (batch, in_height, in_width, in_depth) = in_data.shape
    din_data = np.zeros_like(in_data)
    dweights = np.zeros_like(weights)
    for i_batch in range(batch):
```

```
        for i_height in range(in_height):
            for i_width in range(in_width):
                dout_depth_data = dout_data[i_batch,
                    i_height, i_width, :]
                in_depth_data = in_data[i_batch,
                    i_height, i_width, :]
                dweights += np.dot(in_depth_data.
                    reshape(in_depth_data.size,1),
                    dout_depth_data.reshape
                    (1,dout_depth_data.size))
                din_data[i_batch, i_height, i_width, :]
                    += np.dot(dout_depth_data,
                    weights.T)
    return (dweights, dbiases, din_data)
```

10.6.2　3×3 逐特征图的卷积代码实现

这里卷积核是 3×3。首先，获得 3×3 局部窗口数据，然后乘以权重矩阵（逐元素相乘），得到输出维度为 3×3×depth 的数据，接着对每个 3×3 的数据求和，得到的 depth 个数据即为输出特征图的一个深度维度数据。其代码如下：

```
def dwconv33_layer(in_data, weights, biases, stride=1,
    activation='ReLU'):
    '''
    in_data.shape = [batch,in_height,in_width,in_depth]
    weights.shape = [3, 3, in_depth]
    biases.shape = [1, in_depth]

    out_data.shape = [batch, out_height, out_width, in_depth]
    '''
    (batch, in_height, in_width, in_depth) = in_data.shape
    out_depth = in_depth
    filter_size = 3
    padding = 1
```

```
out_height = (in_height - filter_size + 2*padding)
    //stride + 1
out_width = (in_width - filter_size + 2*padding)
    //stride + 1
out_data = np.zeros((batch, out_height, out_width,
    out_depth))

padding_data = np.zeros((batch, in_height + 2*padding,
    in_width + 2*padding, in_depth) )
padding_data[:, padding : -padding, padding :
    -padding, :] = in_data
height_ef = padding_data.shape[1] - filter_size + 1
width_ef = padding_data.shape[2] - filter_size + 1

for i_batch in range(batch):
    for i_h, i_height in zip(range(out_height),
        range(0, height_ef, stride)):
        for i_w, i_width in zip(range(out_width),
            range(0, width_ef, stride)):
            out_window = weights*padding_data
                [i_batch,  i_height : i_height +
                filter_size, i_width : i_width +
                filter_size, :]
            out_data[i_batch, i_h, i_w, :] =
                np.sum(out_window.reshape(-1,
                out_window.shape[-1]), axis=0,
                keepdims=True)
out_data += biases
# 此处可以加入批量归一化处理代码
if activation != 'None':
    out_data = CnnBlockInterface.activation
        (out_data, activation)
return out_data
```

梯度反向传播过程和正向流程相反，只需对非线性激活和矩阵逐元素乘法进行梯度计算，其代码很简单：

```
def ddwconv33_layer(dout_data, out_data, in_data, weights,
    stride=1, activation='ReLU'):
    '''
    inputs: dout_data, in_data
    outputs: (dweights, dbiases, din_data)
    '''
    if activation !=  'None':
        dout_data = CnnBlockInterface.dactivation
            (dout_data, out_data, activation)
    dbiases = np.sum(dout_data.reshape(-1, dout_data.shape
        [-1]), axis=0, keepdims=True)
    (batch, in_height, in_width, in_depth) = in_data.shape
    (batch, out_height, out_width, out_depth) =
        out_data.shape
    din_data = np.zeros_like(in_data)
    dweights = np.zeros_like(weights)

    filter_size = 3
    padding = 1
    padding_height = in_height + 2*padding
    padding_width = in_width + 2*padding
    padding_data = np.zeros((batch, padding_height,
        padding_width, in_depth) )
    padding_data[:, padding : -padding, padding :
        -padding, :] = in_data
    dpadding_data = np.zeros_like(padding_data)

    height_ef = padding_height - filter_size + 1
    width_ef = padding_width - filter_size + 1
    for i_batch in range(batch):
        for i_h, i_height in zip(range(out_height),
            range(0, height_ef, stride)):
            for i_w, i_width in zip(range(out_width),
                range(0, width_ef, stride)):
                dout_window = dout_data[i_batch, i_h,
                    i_w, :]
                dweights += dout_window*padding_data
                    [i_batch,  i_height : i_height +
```

```
                    filter_size, i_width : i_width +
                    filter_size, :]
                dpadding_data[i_batch,  i_height :
                    i_height + filter_size, i_width :
                    i_width  + filter_size, :] +=
                    dout_window*weights
    din_data = dpadding_data[:,padding:-padding,padding:
        -padding,:]
    return (dweights, dbiases, din_data)
```

通过上面两个算法，我们希望读者对梯度反向传播算法的理解更深刻，能实现一些复杂函数的梯度计算。

10.6.3 逆残差模块的代码实现

逆残差模型的实现很简单，具体如下：

```
def inverted_residual(in_data, weights, biases, out_depth,
    stride=1, expansion_ratio=6, activation='ReLU'):
    out_data11e = conv11_layer(in_data, weights[0], biases[0],
        in_data.shape[3]*expansion_ratio, activation)
    out_datadw33 = dwconv33_layer(out_data11e, weights[1],
        biases[1], stride, activation)
    out_data11 = conv11_layer(out_datadw33, weights[2],
        biases[2], out_depth, activation='None')
    if stride == 1 and out_depth == in_data.shape[3]:
        out_data11 += in_data #  残差模型
    return (out_data11, out_datadw33, out_data11e)
```

梯度反向传播的代码如下：

```
def dinverted_residual(dout_data, out_data, in_data,
    weights, stride=1, activation='ReLU'):
    (out_data11, out_datadw33, out_data11e) = out_data
    if out_data11.shape[3] == in_data.shape[3] and
        out_data11.shape[1] == in_data.shape[1]:
```

```
        out_data11 -= in_data #  残差模型

    (dweights2, dbiases2, dout_datadw33) =
        dconv11_layer(dout_data, out_data11,
        out_datadw33, weights[2], activation = 'None')
    (dweights1, dbiases1, dout_data11e) =
        ddwconv33_layer(dout_datadw33, out_datadw33,
        out_data11e, weights[1], stride, activation)
    (dweights0, dbiases0, din_data) = dconv11_layer
        (dout_data11e, out_data11e, in_data, weights[0],
        activation)
    if out_data11.shape[3] == in_data.shape[3] and
        out_data11.shape[1] == in_data.shape[1]:
        din_data += dout_data #  残差模型
    return ([dweights0, dweights1, dweights2], [dbiases0,
        dbiases1, dbiases2], din_data)
```

10.7　注意机制网络 SENet

在 GoogLeNet 和 ResNet 等网络中，3D 特征图中的每个特征图都是同等重要的，那么是否可以给每个特征图赋予一个权重，表示权重大的特征图更重要，权重小的特征图相对不重要？使网络专注于重要的特征图，这就是注意机制。人类视觉广泛采用了注意机制，我们会特别关注色彩鲜艳和运动中的物体，而忽略其他不重要的内容。SENet 通过挤压和激活模块（Squeeze-and-Excitation）实现这一想法，SENet 是 ImageNet 最后一届（即 2017 年）的冠军网络。

SENet 显式地建模特征图之间的相互依赖关系，通过学习的方式自动获取每个特征图的重要程度，然后依照这个重要程度提升有用的特征并抑制对当前任务用处不大的特征。首先是挤压操作，将每个二

维的特征图变成一个实数，这个实数在某种程度上具有全局感受野，表征在特征图上响应的全局分布，而且使得靠近输入的层也可以获得全局的感受野，这一点在很多任务中都非常有用。SENet 使用全局平均池化作为挤压操作，即把每个特征图的平均值作为实数输出。

其次是激活操作，通过参数 w 来为每个特征图生成权重，其中参数 w 被用来学习显式地建模特征图间的相关性。具体是采用两个全连接层组成一个瓶颈结构来建模特征图间的相关性，并输出与输入特征数量相同的权重。首先，将特征维度通过全连接层降低到输入的 1/16，然后经过 ReLU 激活后再通过一个全连接层升回到原来的维度。相比直接用一个全连接层的好处在于：(1) 具有更多的非线性，可以更好地拟合通道间复杂的相关性；(2) 极大地减少了参数量和计算量，通过一个逻辑函数（Sigmoid）获得 0~1 的归一化权重。

最后是重标定（scale）操作，将激活输出的归一化权重看作每个特征图的重要性，然后通过乘法，逐特征图地加权到先前的特征图上，完成在深度维度上对原始特征图的重标定。

由此可见，挤压和激活模块是利用 3D 特征图的全局信息，通过复杂的全连接学习，得到每个特征图的权重，完成原始特征图的重标定。它是一个独立的模块，可以添加在各种网络结构里。如图 10.3 所示，加载了挤压和激活模块的网络，就称为 SENet。SENet 通过增加很少的参数和计算量，就能显著提高网络的性能。

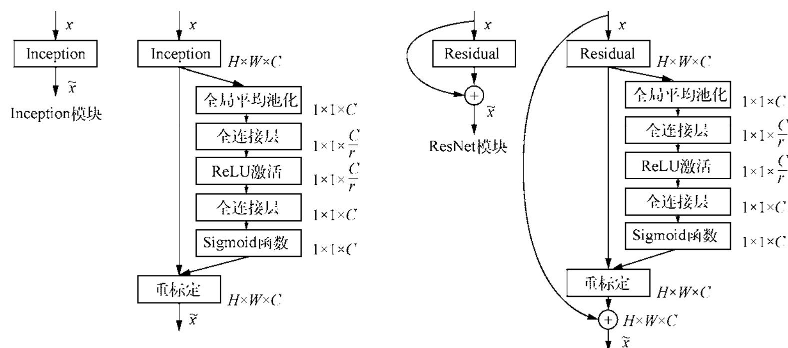


图 10.3　挤压和激活模块加在 Inception 模块（左）和 Residual 模块（右）

第 11 章

工程实践中的问题

通过前面章节的学习，现在可以采集数据集、设计网络结构、选定超参数、训练模型，训练结束后，部署模型，在实践中检验模型效果。如果模型达到预期效果，则皆大欢喜；如果没有达到预期效果，我们该如何提高性能呢？

有很多手段可以改善模型性能：采用不同的网络架构，如 VGG 和 ResNet 等；改变模型容量，如增加网络深度或宽度；尝试不同的优化算法，如 Adam 优化算法；改变超参数，如更改学习率和正则化的强度；尝试批量归一化 BN；更改激活函数，如 ELU；尝试 Dropout；增加训练时间；收集更多数据；收集分布更广的数据。

有如此之多的手段，我们该如何根据实际情况选择呢？为了使模型部署达到预期效果，应该按顺序确保四件事情。

首先，必须确保模型在训练集上得到不错的性能，对于某些应用，可能意味着达到人类的水平。模型在训练集效果不好，主要原因是欠拟合，需要增加模型学习能力。可以采用的手段有采用更好的网络构

架、增大模型容量、更好的优化算法如 Adam、更改学习率和减小正则化强度、采用不同的激活函数如 ELU、增加批量归一化 BN 和增加训练时间。

其次，模型在验证集上也必须有好的表现。模型在训练集效果好而在验证集不好，是因为模型对训练集过拟合了，或者训练集和验证集分布不一致，对应的改善手段有，加大正则化强度或增大训练集。

然后，模型在测试集上也要有好的表现。模型在验证集上性能不错，但测试集不行，这可能意味着模型对验证集过拟合了，你需要往回退一步，使用更大的验证集。模型没有对验证集进行学习，怎么会对验证集过拟合呢？这是因为模型通过验证集来判断各种改善手段的优劣，最终选择验证集效果最好的模型，所以在不断的尝试过程中，会对验证集造成过拟合。

最后，模型部署中必须达到预期效果，使用户满意。如果未达到，这意味着需要退回去，改变验证集、测试集或评估指标。因为如果模型在测试集上做得很好，但在实践中表现不好，这通常意味着测试集分布和实际情况不符，或评估指标设置得不够好。

这四步需要反复迭代，比如在开始时，训练集和验证集获得了满意性能，但测试集效果不理想，此时采用更大的验证集，就有可能导致验证集性能下降，达不到要求，此时又采用增大训练集手段，有可能导致训练集效果变差了，这样又回到第一步，采用各种手段来提高训练集效果。

11.1 单一数字评估指标

如前所述，有很多手段可以改善模型性能，如调整超参数、尝试不同的优化算法或者调整模型架构和规模，如何评估改进后的模型性能呢？这就需要对模型设定性能评估指标，且指标必须是单一数字评估指标。机器学习中，最常用和最简单的评估指标是准确率，即预测正确的样本占总样本的比例，比例越高，说明性能越好。与其相对应的，就是错误率，值为 1 减去准确率。在计算准确率过程中，每个样本都同等重要，这有时不太合适。垃圾邮件检测系统中，模型要判断邮件是否是垃圾邮件，如果是，则邮件放入垃圾邮箱，但当正常邮件被误判为垃圾邮件，则对用户损失可能会很大（比如错过一场重要的会议）；反之，垃圾邮件被误判为正常邮件的损失比较小，用户只需把垃圾邮件移到垃圾邮箱，甚至什么操作都不需要。这样在计算错误率时，正常邮件误判为垃圾邮件的权重要增加，以示其重要性。具体增加多少，根据应用来定。这就是加权错误率，公式为：

$$Error=\frac{1}{\sum w_i}\sum_i w_i I(y_i\neq \hat{y}_i) \tag{11.1}$$

其中，w_i是权重，y_i是样本标签，$\hat{y}_i$是样本预测标签，$I(y_i\neq \hat{y}_i)$是指示函数，当括号内值为真时，函数值为 1，否则为 0。同时模型的损失函数也要采用对应的加权形式。

有时，不仅关心错误率，还要注意模型运行时间，比如需要多长时间来分类一张图片。对时间要求，一般只需运行时间小于一个阈值

（如 50 毫秒），具体是 40 毫秒，还是 20 毫秒，对用户来说没有差别。此时，选择模型，能够最大限度提高准确度，同时满足运行时间要求。

还有一种情况，不仅需要查准率，还需要查全率。在二分类问题中，查准率是指在已经被判断为正样本的样本总数中，有多少是真正的正样本，查全率是指所有正样本被判断为正样本的比例。比如在宝石鉴定中，数据集中有 50 个假宝石，50 个真宝石，模型判断结果为：50 个假宝石中，45 个判断为假宝石，5 个判断为真宝石；50 个真宝石中，48 个判断为真宝石，2 个判断为假宝石，则查准率 P 为 48 /（5 + 48）= 90.6%，查全率 R 为 48/50 = 96%。一般情况下，希望把真宝石都挑出来，即希望查全率越大越好，这样模型就倾向于把样本都判断为真宝石，因此假宝石也易于判断为真宝石，所以查准率会下降。俗语“宁可错杀一千，不可放过一个”，就是指查全率为 100%，但查准率很低；“不可冤枉一个好人”就是指查全率很低，但查准率很高。做到查准率和查全率都很高是比较困难的，查准率和查全率是矛盾的，需要综合考虑，结合查准率 P 和查全率 R 的标准方法是 $F1$ 分数，是它们的调和平均值，公式为：

$$\frac{2}{F1} = \frac{1}{P} + \frac{1}{R} \quad (11.2)$$

对于上面例子，$F1$ 分数等于 93.2%。不同于算术平均，调和平均更重视较小值，取个极端例子，假设 $P = 50\%$，$R = 100\%$，即把所有宝石都判断为真宝石，则 $F1 = 67\%$，而算术平均为 75%。

11.2 人类水平表现

有了单一数字评估指标，通过不断改进模型，指标会越来越好，那么指标有没有一个上限呢？答案是肯定的，模型性能上限就是贝叶斯最小错误率，是理论上可能达到的最小错误率，也就是说，没有任何办法能设计出一个模型，使错误率低于贝叶斯最小错误率。

贝叶斯最小错误率是理论上存在，并不知道具体值。这样对于一个实际任务，因为不知道最小错误率是多少，如果一直优化下去，错误率下降会越来越慢，付出的代价却越来越高，到最后，优化会越来越不划算。模型必须停在某个位置，到了这个位置就不需再优化了。这个位置可以根据实际需求来定，只要满足应用就可以。近年来，随着深度学习研究的深入，取得的突破越来越多，机器学习算法变得很好，这个位置越来越向人类水平靠近。这在深度学习时代前，是不可想象的。为什么机器学习系统要和人类表现进行比较呢？这是因为让机器达到人类水平是自然要求，机器做人类做的事情，代替人类做事情，甚至比人类做得更好。同时，随着深度学习技术的发展，越来越多的特定任务已经达到甚至超过普通人类表现了。

对于人类擅长的任务，如识别图像、听写音频或阅读文字等，人类水平很接近贝叶斯最小错误率。所以对于这类任务，用人类水平作为贝叶斯最小错误率的估计是十分合理的。对人类不擅长的任务，人类水平远差于贝叶斯最小错误率，如从传感器测量数据中判断设备有没有早期故障，要估计贝叶斯最小错误率则十分困难。

为了加深对人类水平表现的理解，以观察医院彩超图像诊断疾病为例。一个未经训练的普通人类，在此任务上可达到 5%的错误率，普通医生达到 1%的错误率，专家错误率为 0.8%，专家团队错误率为 0.5%。此时如何界定人类水平？是 5%、1%、0.8%还是 0.5%?这个问题上，贝叶斯最小错误率可用 0.5%估计，所以人类水平定义为 0.5%更合适。如果希望机器代替人类诊断疾病，机器错误率达到 5%，系统基本不实用；达到 0.8%时，系统比较实用，但不能完全取代专家；如果达到 0.5%，则完全可以取代人类。

事实证明，机器学习性能改善往往相当快，直到超越人类表现之前一直很快，但当超越人类表现时，进展会变慢。这是为什么呢？一个原因是在很多任务中，人类水平接近贝叶斯最小错误率，当系统超越人类表现之后，也许没有太多改善空间了。第二个原因是，只要系统表现比人类表现更差，就可以通过某些手段来提高性能，比如人工标记更多的数据，一旦系统超越了人类表现，这些手段就没那么好用了。比如，不能通过人工标记更多的数据来提高系统性能，因为系统是通过人工标签进行学习的，性能不可能超过人工标记水平。

现在对于很多问题，机器学习已经超越人类水平了。例如网络广告（用户点击广告的可能性）、产品推荐（推荐电影或书籍之类的任务）和预测某人会不会偿还贷款。注意所有这些例子都是从结构化数据中学习的，而且有大量数据可供学习。这些应用中最好的系统看到的数据量可能比任何人类能看到的都多，因此可以比人类更敏锐地识别出数据中的统计规律。对于自然感知任务，如识别图像、语音识别或自然语言处理，人类在自然感知任务中往往表现非常好，机器想要超越

人类会更难一些。但在今天，有些自然感知任务计算机已经超越了人类平均水平，比如一些医疗方面的任务，阅读 ECG 或诊断皮肤癌，或者某些特定领域的放射科读图任务，也许超越了单个人类的水平。

11.3 偏差/方差分析

一旦设定了优化目标，比如人类水平表现，则可以进行偏差 / 方差分析来指导如何改善模型性能。理解偏差和方差的关键在于训练集错误率和验证集错误率。为了方便论述，以图像分类为例，假设人类用肉眼识别几乎不会出错，人类表现的错误率几乎为 0%，则贝叶斯最小错误率可以认为是 0%。

假设训练集错误率是 5%，验证集错误率是 6%。模型对训练集拟合度不高，是欠拟合；相反，对于验证集效果却是合理的，因为验证集错误率只比训练集的多 1%，所以这种是“高偏差”。

假定训练集错误率是 0.5%，接近人类水平表现，验证集错误率是 5%。训练集效果非常好，而验证集效果相对较差，可能过度拟合了训练集，称之为“高方差”。

假设训练集错误率是 5%，偏差相当高；但是验证集的评估结果更糟糕，错误率达到 10%，方差也很高。此时，以降低偏差为主，虽然方差也高。

最后假设训练集错误率是 0.5%，验证集错误率是 0.7%，偏差方

差都很低，是追求的目标。

偏差可以简单地认为是训练集错误率与贝叶斯最小错误率之差，差值越大，偏差越大；方差可以简单地认为是验证集错误率与训练集错误率之差，差值越大，方差越大，一般情况下，验证集错误率大于训练集错误率。

首先看模型的偏差高不高，如果偏差较高，甚至无法拟合训练集，则改善手段有：采用学习容量更大的网络，比如含有更多隐藏层或者隐藏单元的网络、花费更多时间来训练网络、尝试更先进的优化算法、选择更好的超参数等。

其次如果方差高，最好的解决办法就是收集更多数据，其次是加大正则化强度和数据增强来减少过拟合。如果能找到更合适的神经网络架构，有时会同时减少方差和偏差。

一般情况下，我们降低了偏差，方差会相应的提高一些，这是因为偏差低，可能会有一定的过拟合，此时方差会升高。同理，降低了方差，偏差会相应的提高一些，这是因为方差低，可能是由过强的正则化产生的，此时对训练数据的拟合能力降低，偏差会升高。当前的深度学习和大数据时代，只要持续训练一个更大更好的网络，只要准备了更多数据，那么也并非只有上述两种情况。只要正则化适度，一个更大的网络便可以在不影响方差的同时减少偏差，因为容量大的网络学习能力更强，能更好的拟合训练集；采用更多训练数据通常可以在不过多影响偏差的同时减少方差，因为训练数据增多，网络能更好地学习到数据的本质模式，从而提高泛化性能，减小方差。

偏差 / 方差分析指明了优化方向，人类水平表现是贝叶斯最小错误率的有效估计，由此决定是专注于减少偏差还是方差。这个技巧通常很有效，直到系统性能开始超越人类，此时对贝叶斯最小错误率的估计就不再准确了，进展就会变得缓慢。

还以医院的彩超图像诊断疾病为例，错误率 0.5%可以作为贝叶斯最小错误率的估计。当模型超越人类表现时，此时假设模型达到 0.3%训练错误率和 0.4%验证错误率。现在偏差是多少呢？其实很难回答，训练错误率是 0.3%，这是否意味着过拟合了 0.2%（0.5%－0.3%），或者贝叶斯最小错误率其实是 0.1%呢？也许贝叶斯最小错误率是 0.2%？或 0.3%呢？此时没有足够信息来判断优化算法应该专注减少偏差还是减少方差，进展会变慢。所以对于这个任务，一旦系统超过 0.5%门槛，要进一步优化就没有明确选项和前进方向了。

11.4　错误分析

假设模型经过各种优化手段后，验证集错误率还是没有达到要求，此时打算收集更多的数据，收集哪方面数据更有效，能达到事半功倍的效果？此时进行错误分析是最有效的方法，找出验证集中分类错误样本，对错误样本进行归类，针对错误比例较大，且较易改进的错误类别收集更多数据。

假如设计猫分类器，验证集获得了 90%的准确率，这离目标还很远。初略分析验证集分类错误样本，发现将一些狗错分为猫，这些狗

看起来有点像猫，所以可以针对狗的图像优化算法，如收集更多的狗图像，让模型不再将狗错分为猫。问题在于这样做值得吗？最后可能会发现这样做收益很少。错误分析可以快速知道这个方向是否值得努力。

收集比如 100 个错误分类的验证集样本，然后手动检查，一次只看一个，看看验证集里有多少错误标记的样本是狗。假设只有 5 个狗图像，这意味着即使完全解决了狗的问题，也只能修正这 100 个错误中的 5 个。换句话说，如果只有 5%的错误是狗图像，那么错误率最多只能从 10%下降到 9.5%。这称之为性能上限，也就是最好能好到哪里，据此来决定是否花精力解决狗问题。

假设另一种情况，这 100 个错误标记的验证集样本有 50 张图都是狗，这种情况下，如果完全解决了狗的问题，那么错误率就从 10%下降到 5%，效果很好。

错误分析时可以并行评估几个想法，比如改善猫检测器，错误样本可能是由狗或者猫科动物如狮子、豺和猎豹等，或者图像模糊引起的。分别统计这三种错误的比例，假设最后结果是 8%是狗，43%属于猫科动物，61%属于模糊。注意一个错误样本可能有两种错误类型，如模糊的狮子。此时有很多错误来自模糊图片，也有很多错误类型是猫科动物图片。这个结果表明改进模糊图像和猫科动物的潜力最大，知道每种错误类型性能提高的上限空间有多大。具体改善哪类错误，取决于有多少改善性能的手段和投入的人力物力。

总之，通过统计错误类型比例，可以快速发现哪些问题需要优先

解决，或者给出构思新优化方向的灵感，这个分析过程需要经常做。

机器学习中，很鄙视手工操作或者使用了太多人为的数值。但如果要搭建应用系统，这个简单的人工错误分析步骤可以节省大量时间，迅速决定什么是最重要的或者最有希望的方向，这是一个必不可少的过程，强烈建议读者经常进行错误分析。

11.5 修正错误标签

做错误分析时，有时会发现验证集里有些样本被错误标记了，是否花时间去修正这些错误标签呢?

深度学习算法对于训练集中的随机标签错误相当健壮，只要错误样本足够随机，数据集足够大，错误率不太高，模型没有对训练集过拟合，可以不修正这些错误标签。因为同类样本一般都是聚合在一起的，标签错误的样本可以看作其中的孤岛，只要模型没有发生过拟合，分类边界就不能把这些孤岛区域“圈”出来，那么这些孤岛对分类边界基本没有影响，不会影响模型性能。但如果发生了过拟合，分类边界把这些孤岛区域“圈”出来，则这些孤岛对分类边界影响很大，模型性能会急剧下降。当然进行修正也是有价值的。

模型对系统性的错误却很敏感，假如某种类似猫的狗都被错误标记成猫，则分类器学习之后，会把这种狗都错分为猫。因为这些错误标签的样本不再是孤岛，而是大陆，会影响分类边界。

对于验证集或测试集上标记出错的样本，采用错误分析方法，统计标签错误所占的百分比，根据百分比大小判断是否修正。如果标签错误，严重影响了对算法的评估，就要修正错误标签；反之，就可以不修正。

假设验证集中错误标签比例为 1%。验证集有 10%错误率时，此时标签错误只占总错误的 10%（1% / 10%），只是一小部分，所以修正错误标签也许不是当下最重要的任务。但当验证集只有 2%错误率时，此时占到总错误的 50%（1% / 2%），这时修正错误标签是当下最重要的任务。因为标签错误占总错误的比重很大，错误标签对算法整体评估有严重的影响，比如两个模型 A 和 B 在验证集上错误率分别为 2.1%和 1.9%，此时很难判断哪个模型更好了。

如果打算人工修正验证集错误标签，则测试集上错误标签也必须修正，以确保它们来自相同分布。

修正算法判断错误的样本标签容易，修正算法判断正确的样本就很困难。假设算法的准确率为 95%，检查算法判断错误的 5%的样本中哪些标签错误比较容易，但要检查剩下的 95%样本中哪些标签错误比较困难，因为要花很多时间，所以通常不会修正算法判断正确的样本。

通常也不会修正训练集的错误标签，因为训练集太大了，修正要花太多的时间，而且算法对随机标签错误很健壮，修复意义不大。

11.6 训练集和验证集分布不一致

深度学习算法有大量权重，会过早地对小数据集过拟合，导致权重未充分调整，不能泛化到验证集。所以深度学习算法需要大数据支撑，训练集越大算法效果越好。但有时针对某个具体任务能获得的样本有限，网页上却有海量的相似样本，能否利用这些海量样本来提高系统性能呢?

假设开发一个手机应用，判断用户手机拍摄的照片是不是猫。有两个数据来源，一个是真正关心的数据分布，来自用户手机拍摄的照片，当用户数不多时，也许只收集到 1 万张照片，而这些照片一般比较业余，取景不太好甚至很模糊。另一个数据来源是用爬虫程序从网页下载海量猫图，这些图片取景专业、高分辨率、拍摄专业，假设下载数量超过 20 万张。

如果只使用用户手机拍摄的 1 万张照片作为训练模型，效果肯定不好，数据太少了。20 万张网页图片虽不完全来自用户图片分布，但比较接近，对系统肯定有帮助。一般会这样做，训练集包括所有网页图片和部分用户图片，比如 20 万张网页图片和 0.5 万张用户图片；验证集和测试集只有用户图片，比如各 0.25 万张用户图片。这种将数据分成训练集、验证集和测试集方法的好处是，模型如果在测试集达到好效果，则系统肯定就能满足用户需求。缺点在于，训练集和验证集测试集分布不一致，会对偏差 / 方差分析带来困难。

继续用猫分类器为例，贝叶斯最小错误率几乎是 0%，假设训练集错误率是 1%，验证集错误率是 10%。如果验证集和训练集分布一

致，则存在很大方差，算法不能很好地从训练集泛化到验证集。但如果训练和验证数据来自不同分布，就不能得到这个结论。可能是因为训练集容易识别或者训练集都是高分辨率和清晰图片，而验证集是模糊图片，难以识别。所以也许没有方差问题，9%的差别主要反映了验证集比训练集更难识别。模型也可能对训练集过拟合，对与训练集分布一致但未见过的数据的错误率可能是 5%，所以此时方差可能只有5%。为了准确评估方差，首先要准确评估训练集错误率，可以从训练集中随机抽取部分数据，这些数据不用来训练网络，只用来评估网络，功能类似验证集，但这些数据是从训练集抽取的，称之为训练验证集，其分布和训练集一致。这样可以得到分类器在三个数据集的错误率：训练集、训练验证集和验证集。

假设训练集错误率是 1%，训练验证集错误率是 9%，验证集错误率是 10%。这样就能得到结论，模型从训练数据变到训练验证集数据时，错误率上升了很多，算法存在方差问题，因为训练验证集和训练集来自同一分布。

假设训练错误率为 1%，训练验证错误率为 1.5%，验证集错误率上升到 10%。现在方差就很小了，因为从训练数据转到训练验证集数据，错误率只上升了 0.5%。但验证集错误率却很大，所以存在数据不匹配问题。因为学习算法都没有直接在训练验证集和验证集训练过，且这两个数据集来自不同的分布，但算法在训练验证集上做得很好，而验证集上做得不好。如果它们来自同一分布，则错误率应该相同。

再假设训练错误率为 10%，训练验证错误率为 11%，验证集错误

率为 12%。由于贝叶斯错误率大概是 0%，此时存在偏差问题。但基本不存在数据不匹配问题，因为验证集错误率只比训练验证集错误率高一点。

最后假设训练集错误率是 10%，训练验证错误率是 11%，验证错误率是 20%，那么存在两个问题：偏差相当高和数据不匹配问题，但方差很小。

有时会发现验证集错误率比训练验证集低，这说明训练数据比验证集难识别。

总结下，训练集和贝叶斯最小错误率之差为偏差，训练验证集和训练集之差为方差，验证集和训练验证集之差为数据不匹配程度。

如果数据不匹配是模型主要问题，怎么解决呢？很不幸，没有特别系统的方法能解决数据不匹配问题。由于可能存在数据不匹配问题，那为什么还要使用与验证集测试集不同分布的训练数据呢？因为这可以提供更多训练数据，因此有助于提高学习算法的性能。

通过做错误分析尝试了解训练集和验证集的具体差异，为了避免对测试集过拟合，只能对验证集进行错误分析。找到差异后，对训练集数据进行改造使之更像验证集或者采用人工数据合成技术生成训练集。使用这些技巧一定要谨慎，很有可能只是从所有可能的空间中选中了很小一部分去模拟数据。

继续用猫分类器为例，假设发现模糊是主要差异，则可以利用图像模糊技术使训练图像变模糊，更像验证数据。模糊有运动模糊、离

焦模糊等，都可以算法实现。但算法肯定不能完全模拟现实所有的模糊情况，只能模拟很少的情况。

假设研发无人驾驶汽车，利用视觉技术检测车辆，可以利用计算机合成出相当逼真的车辆图像，利用这些合成图像能训练出一个相当不错的系统。不幸的是，合成图像很可能只包含车图像的很小子集，学习算法会对这一小子集过拟合，导致系统对实拍图像性能不佳。

11.7　迁移学习

除了训练集和验证集分布不一致，还有一种情况，就是能获得的样本特别少，甚至网络上都没有相似样本，比如开发识别珍稀鸟类的应用，野外环境下能获得珍稀鸟类的图片很少，可能不到 1 万；医疗中根据彩超判断患者是否患有某种罕见疾病，样本可能更少，可能不到 1 千。这种任务，网络上一般也难以找到相似样本，此时如果直接用这些少量样本从零开始训练深度网络，由于过拟合问题，效果会很差，怎么办？

迁移学习可以解决这类问题，迁移学习是指，任务 A 中学得的知识可以用到任务 B 中，从而降低任务 B 的学习难度。迁移学习对于人类来说，就是举一反三的能力。比如学会了骑自行车，那么学骑摩托车则更容易；学会了旱地轮滑，则学溜冰很容易。这些任务有相似性，知识容易迁移；如果任务之间不具有相似性，则难以迁移，比如围棋知识难以迁移到骑自行车。如果任务之间相似程度较低，则迁移变难，比如围棋知识有可能迁移到象棋。机器学习中的迁移学习和人类类似，

也是希望机器有举一反三的能力。前面两个任务，珍稀鸟类识别或罕见疾病诊断（称为任务 B），都可以用迁移学习来解决，因为它们本质上是图像分类任务，输入是图像，输出是类别。所以可以采用易获得的海量图像，训练深度分类网络（称为任务 A），获得模型 A，任务 A 可以降低任务 B 难度。任务 A 的数据可以从零创建，更常见的方法是采用公开数据集，模型 A 可以重新设计网络结构，然后通过训练得到，也可以采用著名的预训练模型。图像领域最著名的公开数据集就是 ImageNet（含有 1000 个类别的 120 万张图片），采用 ImageNet 的预训练模型如 AlexNet，VGG，GoogLeNet，ResNet 等。

以预训练模型 VGG 为例说明迁移学习具体操作。VGG 模型是多个卷积层和池化层交替，最后接三个全连接层，最后一个全连接层输出 1000 个类别的分值（因为 ImageNet 有 1000 类）。假设任务 B 也是分类，类别有 C 个。迁移学习时，把 VGG 最后一个全连接层换成输出 C 个分值的全连接层，此连接层的权重需要重新随机初始化，然后利用任务 B 的数据对这些随机权重进行训练，注意 VGG 模型的其他层权重冻结，训练时保持不变。由于此时需要训练的参数很少（只有最后一个全连接层的权重），所以即使任务 B 的数据很少，也不容易造成过拟合，能获得好性能。这种迁移学习实质就是把 VGG 模型作为特征提取器，利用任务 B 的数据训练一个线性分类器。因为图像进入 VGG 模型后直到最后一个全连接层，变为一个 4096 维的向量，所以 VGG 模型可以看作把图像变为 4096 维向量的特征提取器。利用这些特征，对最后输出 C 个分值的全连接层进行训练，就是一个线性分类器。从这个角度看，深度学习前特征都是人为设计，如 SIFT 特征，

与任务 B 无关，效果不好。迁移学习特征不是人为设计的，而是由任务 A 决定的，与任务相关，所以效果会更好。特别地，如果任务 A 和 B 相似，效果会更好。比如，珍稀鸟类识别的图像是普通的彩色图像和 ImageNet 图像一样，而罕见疾病诊断的图像是彩超与 ImageNet 图像不一样，所以珍稀鸟类识别与 ImageNet 分类任务更相似。

两个任务相似，不仅要输入数据相似，比如都是图像或语音，还要输出相似，比如都是分类。有时输入相似，输出不同，也可以进行迁移学习，最著名的例子就是物体检测。物体检测输入是图像，输出是物体类别和位置，而图像分类只需输出类别。图像分类得到的模型广泛应用在物体检测中，物体检测算法如 Faster R-CNN 就利用了图像分类得到的预训练模型如 ResNet。

根据任务 B 数据多少和与任务 A 的相似程度，可以把迁移学习简单分为四类。

当任务 B 与任务 A 相似，任务 B 数据很少时，如上所述，只从零开始训练 VGG 模型的最后一个全连接层，其他层冻结。当任务 B 数据较多时，不仅可以从零开始训练 VGG 模型的最后一个全连接层，而且可以对 VGG 模型后面更多层（甚至全部）进行微调，即这些层权重不需重新随机初始化，而是保持原来数值不变，然后利用梯度下降法对这些权重进行微调。注意此时学习率必须低，以避免对原始权重造成太快或太大的改变，经验法则是取训练该模型时学习率的 0.1 倍。

当任务 B 与任务 A 不相似，任务 B 数据很少时，此时很难取得好

效果，可以尝试从 VGG 模型靠前的层开始训练线性分类器。当任务 B 数据较多时，需对 VGG 模型大部分层（甚至全部）进行微调。当然如果数据足够多，也可以从新开始训练一个模型。

迁移学习的理论基础是什么，为什么任务 A 有助于任务 B 呢？以深度学习中图像分类为例，卷积网络通过多个卷积层，逐渐把原始的像素特征变换为边缘、色块等初级特征，然后变换成中级特征如眼睛、嘴巴，最后是高级特征如脸，是分层学习。网络学到的初级特征很通用，能用于各种图像任务中，高级特征跟任务密切相关，通用性差。比如 ImageNet 中包含大量的狗种类，所以高级特征中很多属性可能用于区分不同的狗品种。由于初级特征通用性好，所以任务 A 学到初级特征能用于任务 B，具有迁移性。如果任务很相似，高级特征也能互相迁移；不相似时，高级特征难以迁移。深度学习中分层学习，初级特征的通用性是迁移学习的基本理论。

迁移学习在实践中大量使用，很多任务中训练数据很少，所以可以找一个数据很多的相似任务来预先学习，并将知识迁移到这个新任务上。

迁移学习也大量用在模拟中，对很多依靠硬件交互的机器学习应用而言，在物理世界中收集数据、训练模型，要么昂贵，要么耗时间，要么危险，所以最好能以其他方式来收集数据，计算机模拟是首选工具，从虚拟世界中学习并将学到的知识迁移到物理世界。比如无人驾驶，真实道路下进行无人驾驶训练，太危险也十分耗时，OpenAI 的 Universe 平台用视频游戏（如飞车游戏）来训练无人驾驶汽车。又比

如训练机器臂抓取物体，在物理机器人上训练模型非常缓慢和昂贵，但计算机模拟训练则快速又廉价。

11.8 多任务学习

迁移学习是任务 B 向任务 A 学习，不存在反向学习，即任务 A 向任务 B 学习。多任务学习是双向学习，任务 A 和任务 B 互相学习，甚至数个任务之间互相学习。给定多个学习任务，其中所有或一部分任务是相关的但并不完全一样，多任务学习的目标是通过使用这多个任务中包含的知识来帮助提升各个任务的性能。迁移学习时，任务 A 拥有海量数据，任务 B 数据很少，只能先训练任务 A 得到模型 A，训练任务 B 时，只是对模型 A 进行微调。多任务学习时，所有的任务同时训练，没有先后之分。多任务学习的理论基础和迁移学习一致，任务获得的知识有助于其他相似任务学习。多任务训练时，各个任务同步学习，各自学习任务相关知识，同时所有任务共享这些知识，降低了学习难度，提高了泛化能力。多任务学习广泛存在人类社会，学生在学校学习多门课程和交叉科学研究都是多任务学习，“他山之石，可以攻玉”说的也是多任务学习。

深度学习时代，多任务学习理论基础和迁移学习一致，即深度学习中分层学习理论，网络底层学到的特征通用性好，能用于不同任务中。多标签学习是一种典型的多任务学习，以多标签学习来具体说明多任务学习过程。ImageNet 是单标签学习，即一张图片只有一个标签，是一种单任务学习。多标签学习是指一张图片同时有多个标签，模型

需要同时输出这些标签，每个标签对应一个任务。比如研发无人驾驶车辆，需要对拍摄的道路图像判断是否有行人、车辆、交通标志还有交通灯等。单标签学习时，模型是多个隐含层后接一个输出层，然后计算 softmax 损失，进行误差反向传播更新权重。多标签学习时，模型是多个隐含层后接多个输出层（每个输出层对应一个标签），然后分别计算每个输出层（标签）的 softmax 损失，每个损失都进行误差反向传播更新权重，可见多个隐含层的权重被每个标签损失更新。多标签学习时，多个输出层共享多个隐含层（共享网络），每个损失都对共享网络同时进行训练。每个标签任务不同之处在于都有独立的一个输出层，其权重由各自损失进行独立更新，以适应不同标签任务。换个角度看，共享网络可以看作是多标签学习的特征提取器，每个标签任务都共享该特征提取器，具有同样的特征表示，并独自训练一个线性分类器，共享网络是所有任务共同训练出来的。

当然，也可以训练多个网络，每个网络只解决一个标签任务，训练一个共享网络完成多个任务会比训练多个完全独立的网络分别完成一个任务，性能要更好，这就是多任务学习的力量。多任务学习可以提高网络的泛化性能、加快学习速度和减少达到单任务同等性能水平所需要的训练样本数量。

为什么多任务学习性能比多个独立任务要好，有多种可能的原因。多任务学习允许共享网络中专用于某个任务的特征被其他任务使用；共享网络可以学习到可适用于不同任务的特征，这样的特征在单任务学习网络中往往不容易学到；对于某个任务，误差反向传播时其他任务产生的梯度可以视为噪声，这些噪声可以提高该任务的泛化能

力。

多标签学习标记样本时，感觉每个样本要标记所有标签，事实证明，多标签学习也可以处理样本部分标记的情况。比如有的样本标记了有车，但忘了标记是否有行人；或者有的样本标记了没有行人，没有标记是否有交通灯等等。即使是这样的数据集，也可以进行多标签学习，训练网络时，不是每个标签都进行反向传播更新权重，只对有标记的标签进行权重更新，忽略没有标记的标签。

多标签学习时，任务很相似，只有输出层不共享，其他层都共享。如果每个任务不太相同，则可以增加不共享的隐含层数，每个任务学习到更多特有的特征，来提高学习效果。物体检测是任务不相似的多任务学习实例，物体检测输入图像，输出多个物体的类别和位置，检测每个物体就是一个任务，判断物体的类别和位置是两个子任务，性能完全不同，所以物体检测网络中判断类别和位置的子网络采用了不同的网络结构，同时损失函数也不同，类别采用 softmax 损失，位置采用 L2 范数损失（均方损失），详细内容可参考 Faster R-CNN 等网络。

有时，单任务学习可以通过增加一些辅助任务来提高主任务的学习效果，进行多任务学习。以人脸关键点定位为例，它需要精确定位眼睛、鼻子和嘴角位置，很容易受遮挡和脸部姿势的影响，如正脸和侧脸差别极大。通过设计一些辅助任务来提高主任务效果，这些辅助任务与主任务密切相关，比如一个正在笑的孩子会张开嘴，有效地发现和利用这个相关的脸部属性将帮助更准确地检测嘴角。通过设计是否带眼镜、笑脸、性别和脸部姿态这 4 个辅助任务，显著提高了人脸

关键点定位精度。

多任务学习中每个任务都对共享网络权重进行更新，更新比例由任务重要性决定，任务越重要，比例越大；任务相似，则比例相近。比如多标签学习中每个标签的比例可以取相同，辅助任务学习时，主任务比例更大。当任务采用不同的损失函数时，则还需对损失归一化。

最后强调一点，多任务学习时，如果各个任务很相似，则多任务学习优势很大；如果任务不相似，则效果会变差，特别的，离群任务会显著降低系统性能。如果任务相似，多任务学习会降低性能的唯一情况，和训练多个单任务相比，就是神经网络容量不够大。

11.9 端到端学习

深度网络强大的拟合能力和大数据支撑，兴起了端到端学习，端到端学习是采用海量训练数据用巨大网络直接学习从输入到输出的复杂映射关系。传统学习方法，中间会有多个人工处理阶段，端到端学习不需要进行这些处理。

深度学习的图像分类可以看作一种端到端学习，从图像直接输出类别，中间没有任何人工处理；传统方法需要先提取人工设计的特征（如 SIFT 特征），然后输入到分类器（如 SVM 分类器），中间有人工特征阶段。

深度学习的语音识别也是端到端学习，训练一个巨大的神经网络，输入是一段音频，输出是对应的听写文本，中间不需要任何人为

设计的特征。传统上语音识别是多阶段的方法，首先提取一些手工设计的音频特征，比如著名的 MFCC 特征。提取出这些低层次特征之后，应用机器学习算法在音频片段中定位音位，音位是声音的基本单位，比如说“Cat”这个词是三个音位构成的：Cu、Ah 和 Tu，算法把这三个音位提取出来，然后将音位串在一起构成独立的词，然后将词串起来构成音频片段的听写文本。

再比如机器翻译，传统上机器翻译系统也是一个很复杂的流水线，比如英语翻译到法文，先做文本分析，从文本中提取一些特征，然后经过很多步骤，最后翻译成法文。现在广泛采用端到端学习的机器翻译，输入英文，网络直接输出法文。

为什么现在大量采用端到端学习，放弃手工设计特征呢？因为有了海量数据，加上深度网络强大的拟合能力，端到端学习效果高于手工方法。但是必须指出，当数据集较小的时候，传统方法效果也不错，通常做得更好。

端到端学习的最大优点是只让数据说话。如果有足够多的训练数据，直接训练一个足够大的神经网络，纯粹使用机器学习方法，捕获数据中的统计信息，而不是被迫引入人类的成见（有时有帮助，有时没有帮助）。第二个好处就是手工设计组件更少，所以也许能够简化设计工作流程，节省时间。第三个好处是能减小样本标记工作量，端到端学习只需标记最后输出标签，通常是比较简单的。传统方法需要标记很多中间过程而且标签很复杂。

端到端学习的缺点也是明显的，首先可能需要大量的数据。如果

没有大量数据，性能可能没有传统方法好。另一个缺点是，排除了可能有用的手工设计组件。手工设计组件可以把人类知识直接注入到算法，引入先验知识，有时会提高系统性能。学习算法有两个主要的知识来源，一个是数据，另一个是先验知识，所以当有大量数据时，先验知识不太重要，但是当没有太多数据时，先验知识对系统很有帮助。

由于端到端学习上述的优缺点，对于一个实际问题，需要对问题进行拆分，把一个复杂问题拆分成一系列比较简单的问题，对每个简单问题再通过端到端学习解决，尽量少的引入手工设计组件，复杂问题的拆分是一门艺术。通常来说，在识别、检测和分割等基础任务上，端到端学习可以获得更好的效果。

比如无人驾驶，车辆安装有各种传感器，如摄像头、毫米波雷达、激光雷达等，这些传感器采集到的时序数据是输入，输出是刹车、油门和方向盘的操作序列。如果直接用巨大的网络拟合这些输入输出，就是端到端学习。

目前端到端学习可能不适合开发实用无人驾驶系统，不如拆分为感知、地图、决策三个部分。其原因如下：第一，不聪明。在做驾驶决策时，只关心高精地图环境、本车当前位置和周围物体的相对位置，并不关心物体的颜色，如车的颜色或者路边树叶是绿的还是黄的。端到端学习没有这些先验知识，所以需要大量冗余数据和计算。对于拆分方案，分别独立学习再融合，每个部分可以采用端到端学习，可以大大降低需要的数据和计算。第二，不灵活。如果传感器位置、性能变化或者增加其他感知设备，就可能需要重新收集数据学习或进行高

精度的传感器校正。如果换辆车，执行机构变化，也需要重新收集数据学习。拆分方案可以大大提高灵活性。第三，难理解。无人驾驶是一个系统工程，遇到问题时，需深入系统诊断出问题模块，有针对性的改进，是解决问题的行之有效的手段。但是对于整体端到端学习，一旦出现问题，因为无法“对症下药”，解决问题的难度会增大，需要投入更多的资源和时间。拆分方案十分容易理解，排查问题很容易。第四，多解性。面对相同的道路环境，每个驾驶员的操作是不同的，所以端到端学习时同一个输入会对应多个输出，造成拟合困难。拆分方案中的决策部分可以很好的处理这个多解问题。

但是，并不能因此完全否定端到端学习，这只是端到端学习运用到无人驾驶领域目前所存在的问题，或许在将来可以得到解决。但是目前，对于能收集到的数据、能够用神经网络学习的数据类型以及训练神经网络的方法，端到端学习实际上不是最有希望的方法。

现在高铁进站闸机，采用人脸票证合一自助进站，快捷方便。旅客站在闸机前，身份证放在闸机上，如果相机识别出的旅客身份和机器读取身份证的信息一致，则旅客可以通过闸机。如何设计身份识别系统呢？如果采用端到端学习，则系统直接学习图像到人物身份的函数映射，事实证明目前这不是最好的方法。相反，迄今为止最好的方法似乎也是拆分方法：先检测后识别。首先，运行人脸检测算法定位人脸，然后缩放人脸图像并裁剪图像使人脸居中，再用神经网络识别身份。训练识别网络的方式是输入两张图片（一张是身份证照片，一张是裁剪后的人脸），判断是否是同一个人。如果身份信息不一致，则会进一步与公安系统所有在逃犯进行一一比对，也是输入两张图片(一

张是在逃犯照片，一张是裁剪后的人脸），如果判断为同一人，则系统报警。为什么两步法更好呢？有两个原因。一是这两个子任务比较简单，二是两个子任务的训练数据都很多，两个任务都可以用端到端学习。相比之下，如果想一步到位，从拍摄照片到身份证号码，这样的数据对就少得多。注意这种拆分不需先验知识。

再如，利用摄像头进行驾驶员疲劳检测，输入是驾驶员照片，输出是否疲劳。如果采用端到端学习，则很难采集到大量疲劳照片。采用拆分方法更简单——先进行人脸关键点定位再判断是否疲劳。人脸关键点定位有大量数据且解决得很好，人脸关键点定位定位出眼睛，嘴巴等，再根据人疲劳时眼睛和嘴巴状态等先验知识，如疲劳时会频繁眨眼，嘴巴打哈欠等，进行疲劳判断。具体的，输入是眼睛图像，输出是分类（闭眼还是睁眼），然后依据疲劳理论，根据闭眼睁眼时序结构来判断是否疲劳。注意这种拆分利用了先验知识。

实际中是否采用端到端学习，关键问题是判断是否有足够的数据学到输入到输出的复杂映射。如果有足够数据，可以尝试端到端学习；如果不多，可以尝试拆分为几个子问题，每个子问题采用端到端学习，同时尽可能少使用先验知识，如有必要，最好使用简单明了的先验知识；如果数据很少，还是采用传统方法。

11.10 修改评估指标或者验证集测试集

评估指标是用来评估模型优劣的，假设评估指标采用分类准确率，现在有两个模型 A 和 B，在测试集上错误率分别是 3%和 5%，可见模

型 A 性能更好。但模型实际部署后，用户发现模型 B 更好，而不是模型 A。

一种可能原因是，模型 A 把一些重要的样本判断错误，而模型 B 没有，对用户来说，这些重要样本不能判断错误，所以用户更喜欢模型 B。这时原来的评估指标已经无法正确评估算法的优劣，需要定义一个新的评估指标。新指标要突出重要样本，一种可能方式就是 11.1 节中定义的加权错误率。新评估指标的意义在于准确指出两个模型哪一个更适合应用。

还有一种原因是，测试集和用户使用模型时的数据分布不一致，比如测试集都是高质量图像，而用户图像比较模糊，巧合模型 B 在模糊图像性能更好。此时就需要修改验证集测试集，使之与用户数据分布更一致，更能反映实际需要处理的数据。

好的评估指标和测试集可以更快做出决策，判断模型 A 还是模型 B 更优，加速模型迭代速度。即使无法定义出一个完美的评估指标和测试集，先快速设立出来，然后使用它们来驱动模型迭代。如果在这之后，发现有更好的想法，可以立即修改，最好不要在没有评估指标和测试集时耽搁太久，因为可能会减慢模型迭代速度。

11.11　如何设计训练集、验证集和测试集

如前所述，希望这三个数据集来自同一分布，所以建议把所有数据随机洗牌，然后按一定比例分割为三个数据集，验证集测试集必须

和系统部署后用户的数据分布一致。

深度学习时代前，有一条经验法则把全部数据用 70/30 比例分成训练集和测试集，或者如果必须设立训练集、验证集和测试集，则按 60/20/20 比例划分。这样划分是相当合理的，因为数据集都比较小。

但在现代机器学习中，数据集变得很大，比如说有 1 百万个样本，这样划分可能更合理，98%作为训练集，1%验证集，1%测试集。因为如果有 1 百万个样本，那么 1%就是 1 万个样本，这对于验证集和测试集来说可能已经够了。因为验证集是用来分辨不同模型的好坏，只要验证集足够大，能够准确区分模型好坏就可。测试集的目的是完成系统开发之后，测试集可以评估投产系统的性能，只要测试集足够大，能够以高置信度评估系统整体性能就可。这样就留有更多的样本用来训练模型，满足深度学习算法对数据的大胃口。

传统机器学习一般用交叉验证法对模型进行评估，即所有数据只划分为训练集和测试集，然后把训练集随机划分为相等的 k 个子集。每次取 k-1 个子集进行训练，剩下的那个子集作为验证集；这样就可获得 k 组训练集 / 验证集，进行 k 次训练和验证，最终取 k 次验证结果的平均值。因为传统时代，数据集很小，验证集也很小，只进行一次验证会导致很大方差，结果十分不准确，采用交叉验证法能比较可靠地评估模型。深度学习一般不需采用交叉验证法，因为验证集足够大，不会产生大方差，评估模型很稳定。

11.12　类别不平衡

一般数据集中各类样本的数量相当,如果各类样本数量稍有差别,通常影响不大。但当差别很大时，会造成很大困难，这称为类别不平衡问题。实际中这种情况很多，比如欺诈预测（世界还是好人多），自然灾害预测（地球是仁慈的），识别恶性肿瘤（不幸的人毕竟是少数）等。假设二分类问题，正类数量是 0.1 万，负类数量是 100 万，如果模型只是把新样本永远预测为负类，就能达到 99.9%的准确率，但其实模型什么都没有学习，没有任何价值，因为不能预测出正类，所以此时不能用准确率来评估模型。而要采用 *F1* 分数的加权形式 F_β，表达出对查全率查准率不同偏好，定义为

$$\frac{1}{F_\beta}=\frac{1}{(1+\beta^2)}\left(\frac{1}{P}+\frac{\beta^2}{R}\right) \tag{11.3}$$

其中 β 等于 1 时是 *F1* 分数，β 大于 1 时偏重查全率，β 小于 1 时偏重查准率。类别不平衡问题一般对稀少类偏重查全率，可以牺牲一定的查准率,例如识别恶性肿瘤时肯定要把所有肿瘤都识别出来，不能把肿瘤识别成正常组织，因为患者可能因此失去最佳治疗时间；相反把正常组织识别成肿瘤，则代价很小。可令 β 等于 5，比如，数据集中肿瘤样本 10 个（正类），非肿瘤样本 10000（负类）。如果模型把样本都预测为非肿瘤样本，则查全率为 0，查准率为 0，F_β为 0，准确率却为 99.9%，显然 F_β合理很多。假如模型为了不漏测正样本，把所有正类都预测为正类，但同时负类预测为正类的比例也会增加，如为 1%，则查全率为 100%，查准率为 10/(10+10000*1%)=9.1%，F_β为

72.2%，$F1$ 为 16.7%，可见 F_β 更偏向查全率。准确率为 99.0%，还是 F_β 合理。只有模型把正负样本都分类很准确，F_β 才能很高。这个例子对生活很有启示，对于一些发病率很低的严重疾病，此时正样本比例很少，检测仪器为了提高查全率，往往查准率很低，可能低于 10%，所以即使检测为阳性，不要太担心，因为查准率很低，大部分都是假阳性，但是进一步的排查还是很有必要的。

类别不平衡问题之所以会对机器学习方法造成问题，是因为模型的损失函数定义方式与评估指标 F_β 不一致造成的。损失函数是对一个批次的每个训练样本的损失取算术平均，这种定义方式中每个样本的权重都是一样的，此时用准确率来评估模型是最合适的。但如前所述，准确率用来评估类别不平衡问题是十分糟糕的，应采用 F_β 评估。但目前的问题是，与 F_β 评估一致的损失函数还没有被发现，或者太复杂且不可导，以至于不能使用梯度下降法进行训练，没有实际价值。只能退而求其次，找到一种损失函数能较好近似 F_β 指标，目前最常用最简单的是加权损失函数，即对正负样本的损失取不同权重，一般是增加稀少类别的权重，因为稀少类别一般更重要。那么正负样本权值如何取呢？一般来说，根据真实生产环境中的正负样本比例来取值较为稳妥。

还有一种损失函数是所谓的 FL（focal loss）损失，也是加权损失，只是权重不是常数，而是对训练好的样本权重小。因为类别不平衡时，多数类很容易训练到较高概率，少数类很难训练好，所以如果能自适应地减小高概率样本的权重，则可以减轻多数类的影响，FL 损失公式如下：

$$FL(p_t)=-(1-p_t)^{\gamma}\log(p_t) \quad (11.4)$$

其中 p_t 是样本的概率，γ 是大于 0 的参数，通常取 2。可见与常规的损失函数相比，权重为 $(1-pt)^{\gamma}$，这样当 pt 大于 0.6 时，权重很小；pt 小于 0.3 时，权重较大。这样可以让损失聚焦于难样本（即少数类样本），提高其学习效果。

损失函数可以综合上面两种类型，得到：

$$FL(p_t)=-a_t(1-p_t)^{\gamma}\log(p_t) \quad (11.5)$$

at 是常数权重，对于多数类取值 0.75，少数类取 0.25 比较合适。

采用加权损失函数后，如果模型效果还不行，优先考虑采集更多的数据以增加稀少类别的样本数。有两种方式，第一是只增加稀有类别的样本数，多数类别样本不增加，这种方式增加的样本总数少，节省计算资源，能显著提高稀有类别的查全率，故常采用这种方式。第二种就是按原始比例增加所有类别样本数据，这种方式增加的样本总数很多，需要大量计算资源。

然后再考虑对训练集进行过采样或欠采样处理，优先考虑过采样。

1、过采样，人为增加数量少类别的样本。一种是直接复制样本，另一种是对样本进行改造生成“新样本”。深度学习采用数据增加技术来增加样本，但增加比例不能过高，如一个样本最多生成 20 个样本。如果比例过大，会对稀少类别过拟合，增大方差。

2、欠采样，删除样本数量多的类。一般希望删除很容易区分的样本，让算法专注于难区分样本，这样模型的泛化性能更好。有三种方式删除样本，一是随机删除；二是利用先验知识人为删除易区分样本，比如删除与肿瘤图像明显不同的图像；三是先训练一个模型，然后根据模型结果删除易区分样本，比如采用 softmax 分类器时，如果样本的概率很接近 1，说明样本很像该类别，易于区分，如果概率很低说明特征不明显，模棱两可，所以可以删除概率接近 1 的样本。这三种方法可以单独使用，也可以同时使用，即首先人工删除易区分样本，然后对数量多的类别下采样，使各个类别数量相当，训练一个模型，使用模型删除概率很高的样本，最后如果还不平衡则随机删除。其中模型删除法比较复杂，读者可以查阅相关文献。欠采样由于删除了部分样本，这部分样本有可能会预测错误，所以查准率一般会提高，查全率有可能也会提高，这样就造成了对实际系统的乐观估计，所以需要谨慎采用。

当类别不平衡很严重时，这两种方法可以同时采用，对少样本的类别过采样，对多样本的类别欠采样。

经过采样后，训练集中稀少类别的比例变大，此时数据分布与实际情况不一致了，会产生数据不匹配问题。如果数据不匹配问题很严重，则采样后的比例尽可能保持原有比例，如果不严重，比例可以向 1:1 靠拢，以降低网络训练难度。具体比例需要平衡数据不匹配问题和网络训练难度。

验证集测试集不能进行采样处理，只能包含原始样本，需与实际

部署情况一致，此时需用 F_β 指标评估模型。

采用随机梯度下降法训练模型时，由于损失函数是加权的，每个批量样本中不需要每类样本数量一样多，只需和训练集的比例一致即可。

由于稀有类别的样本数一般较少，多采用迁移学习。如果稀有类别的样本数很少，也可以采用单分类（one-class）的异常检测方法。

最后还可采用集成学习方法，其思路是采用随机欠采样方法减少各大类中的样本数量，得到相对平衡的数据集，学习得到一个分类器；多次独立地重复以上采样学习过程，得到多个分类器；集成所有分类器得到最终模型。集成方式常用随机森林或 AdaBoost 方法。

11.13　负样本采集

二分类问题中，正类就是要检测的类别，负类就是非正类或称为背景。假设研究无人驾驶，要设计车辆分类器，此时正类就是车辆，负类就是没有车辆的背景区域。首先要收集样本，就是要收集大量正负样本，且数量大致相当。正样本好收集，就是收集各种包含车辆的区域，那负样本呢？感觉就是收集大量不包含车辆的区域，其实并不简单，是机器学习里面一个比较难解决的问题。因为不包含车辆的区域太广泛了太多了，比如天空海洋沙漠行人建筑路面树木等等，如果任意选择势必会造成负样本过量，导致类别不平衡问题，同时有可能会选出很多与车辆差别很明显的区域，导致分类器很容易学习出，但

分类器性能比较差，因为遇到与车辆比较接近的负样本就容易预测错误。所以负样本采集的核心问题就是采集尽可能少的负样本，同时训练出性能好的分类器，关键是采集到难样本，即与车辆很相似的负样本。

为了减小采集样本的数量，一般只在限定场景采集，比如无人驾驶只需在道路场景采集，不需要在天空海洋沙漠等场景采集，因为无人车不可能采集到这些场景的照片，所以无需对这些照片进行区分，这样就大大缩小了范围，减小了负样本数量。

为了采集到难样本，一般采用迭代法，逐步采集到越来越难的样本。首先在限定场景内收集海量不包含正样本的负样本集，然后在每张照片中随机截取数个区域（因为车辆一般是整幅照片的一个区域，所以负样本也是图片的一个区域），一张照片可以只截取一个区域，也可以截取多个区域。这些区域组成首轮负样本集，其数量和正样本一样多，或 2 倍或 3 倍。利用这些负样本和正样本训练分类器。有了分类器，就可以利用分类器采集难样本，即分类器预测正确的负样本是易样本，预测错误的是难样本。第二轮负样本集采集方式为：分类器首先对首轮负样本集进行预测，如果预测为正样本，则放入第二轮负样本集；其次分类器预测海量负样本集中随机区域，如果预测为正样本，则放入第二轮负样本集，直到第二轮负样本集数量达到要求。再利用第二轮负样本集和正样本训练分类器，又利用此分类器采集第三轮样本集，采集方法不变，再训练，如此不断迭代，选出的负样本会越来越难，分类器性能会越来越好，分类器的学习容量也因此需要越来越大。后期挑选出负样本会很难，需要从大量区域去挑选，需要大

量计算资源，比如后期分类器性能很好，负样本错分率为 1%，则要产生 1 万的负样本，需要大概采集 100 万个区域，并对这 100 万个区域进行分类。

最终模型可以是最后一轮输出的复杂模型，有可能需要使用所有轮收集的负样本和正样本对该模型进行微调，采用加权损失函数，即后面轮的负样本权重越来越高，因为样本越来越难，正样本权重最大。

最终模型也可以是所有轮输出模型的集成，此时不需要微调模型，样本依次进入各轮模型，如果某轮模型预测为负样本，则预测结束；如果预测为正样本，则进入下一轮模型，直到最后一轮模型。集成模型的优点是预测速度快，因为实际场景中，大部分样本是易识别的负样本，因此只需很少轮的预测，就能预测为负样本，从而结束预测过程；只有极少的难负样本需要多轮预测，注意正样本需要到最后一轮，但正样本在实际场景中的数量很稀少。且前几轮模型可以设计得很轻量，如线性模型或者只有一个隐含层且单元数很少的模型，预测速度很快。还可以采用的加速技术有：前几轮模型不需要样本所有特征，只需样本最具有区分度的几个重要特征，后面的几轮才取越来越多的特征。具体实现方式读者可以阅读著名的基于 AdaBoost 的人脸检测算法。

其实多分类问题也存在负样本采集问题，假设无人驾驶，要设计商务车、乘用车、公交车、卡车、三轮车、摩托车多分类器，道路场景中的背景区域就是这些类别的共同负类，只需把要检测的所有类别看作一个超级正类（都是各种车辆）就好理解。采集负样本的手段和

二分类一致，也是采用迭代法采集越来越难的样本。这项技术广泛运用于目标检测领域。

11.14 快速搭建并迭代系统

如果开发全新的机器学习应用，应该尽快建立第一个系统原型，然后快速迭代。

假如建立一个新的语音识别系统，可以优先考虑很多事情。比如有一些技术对嘈杂背景更加健壮，如咖啡店噪音，里面有背景音乐或很多人聊天，或者车辆噪音。有一些方法对口音更健壮，还有所谓的远场语音识别问题即麦克风与说话人距离很远。儿童语音识别带来特殊的挑战，挑战来自单词发音方面，还有他们使用的词汇。还有说话人口吃或者说了很多无意义的短语，比如“哦”,“啊”之类的。可以选择很多不同的技术，让听写下来的文本可读性更强，可以做很多事情来改进语音识别系统。

一般来说，几乎所有的机器学习应用都可能会有 50 个不同的方向可以前进，并且每个方向都是相对合理的。但挑战在于，如何选择一个方向集中精力处理。所以笔者建议，如果搭建全新的机器学习应用，不要把系统设计得太复杂，要快速搭出第一个粗糙模型，然后开始迭代。就是快速设立验证集、测试集和评估指标，这样就决定了系统的目标，然后找到训练集，搭好一个机器学习系统原型，训练一下看看效果，开始理解算法表现，在验证集测试集评估指标上表现如何。

采用偏差 / 方差分析和错误分析，来确定下一步优先做什么。特别地，如果错误分析表明大部分错误来源是说话人远离麦克风，那么就集中精力研究远场语音识别技术。即使最初的目标定错了，之后也是可以修改的。

第一个粗糙模型尽量做到对训练集效果很好，即使过拟合也可以。构建原则为，网络架构采用经典结构如 VGG 或 ResNet，降低正则化强度甚至可以不要正则化，优化算法采用 Adam（使用默认参数），学习率取默认值 0.001，激活函数采用 ReLU，评估指标采用准确率，然后在满足运行时间前提下尽可能采用大容量网络，以加大网络深度优先。接着只对学习率超参数进行调优，使模型对训练集效果很好，如果模型对训练集效果始终不是很好，尝试把激活函数改为 ELU、增加批量归一化 BN。当模型对训练集效果很好时，如果此时有较大方差，慢慢提高正则化强度或增加 Dropout，减小过拟合。之后才能采用错误分析确定下一步优先方向。

如果读者在某个应用领域有很多经验，建议适用程度要低一些。还有一种情况的适应程度更低，当这个领域有很多学术文献时，那么可以借鉴学术文献，一开始就搭建比较复杂的系统，比如人脸识别、人体关键点定位和人脸关键点定位等等。

第 12 章

目标检测

图像分类是判断图像中是否有感兴趣的对象（如车辆），这是计算机视觉的基础和核心任务。图像分类的用处比较有限，比如在无人驾驶中，只知道前方道路有车辆是不够的，还需知道车辆的位置，才能采取合理的驾驶策略，如减速、加速或跟车。目标检测就是用来解决这个问题的。

12.1　目标定位



图 12.1 目标定位

为了让大家更清晰地理解概念，先从最简单的定位任务开始。在图像分类任务中，通常图像只有一个对象且对象较大，占据了大部分面积，分类任务需要给出对象的类别，如汽车、摩托车和行人等。目标定位任务和分类任务十分相似，图像中也是只有一个较大的对象，但需要给出其类别和位置。表示物体的位置，目前最主流的做法是用一个水平矩形框包围物体，矩形框要能全部包围物体且面积最小，即要求矩形框尽可能接近物体边界，该矩形框称为边界框（Bounding Box），所以只要确定了边界框的位置就相当于定位了物体。在图像的二维平面上，描述边界框需要 4 个参数，最常用的方式是给出边界框的中心坐标（bx, by）和高度宽度（bh, bw），这 4 个参数[bx, by, bh, bw]称为边界框向量。为了便于网络学习，这 4 个元素需在 0 到 1 之间，所以需要对图像坐标进行归一化，即定义图像左上角像素坐标为（0, 0），右下角像素坐标为（1, 1）。

图像分类任务是通过端到端学习的，输入图像到多层卷积网络，网络输出分值向量，最后由 softmax 层预测图像类别。那么如何得到边界框向量呢？能不能采用多任务学习思想，让网络不仅输出分值向量，同时还输出边界框向量呢？这完全是可行的，而且实践证明效果很好。目标定位的核心思想是端到端学习和多任务学习，其实目标检测的核心思想也一样是这两点，如果读者对这两种学习方法不熟悉，希望先看第 11 章，再继续下面的学习。

以无人驾驶为例，为了便于描述，简化问题，假设只检测汽车、摩托车和行人 3 类对象，因为道路中有时只有背景，并没有检测对象，所以网络输出向量应该是 $\boldsymbol{y}=[po, c1, c2, c3, bx, by, bh, bw]$8 维向量，而

分类只需输出 $\boldsymbol{y}$=[*po*, *c*1, *c*2, *c*3]4 维向量。制作训练样本的标签时，8 维向量具体取值如下，如果图像中有检测对象（汽车、摩托车或行人），则 *po*=1；如果图像中只有背景，则 *po*=0；[c1, c2, c3]是分值向量，目标定位问题中最多只有一个对象，所以 *c*1，*c*2，*c*3 中有且仅有一个等于 1，其他为 0。比如图像中有汽车，则 *c*1=1，*c*2=0，*c*3=0，有摩托车则 *c*1=0，*c*2=1，*c*3=0，有行人则 *c*1=0，*c*2=0，*c*3=1；[*bx*, *by*, *bh*, *bw*]是对象边界框向量。当 *po*=0 时，由于没有对象，所以剩下的 7 个元素毫无意义，不需要指定。

定义损失函数时，*po* 采用逻辑回归函数（Logistic regression），[*c*1, *c*2, *c*3]采用 softmax 损失函数，[*bx*, *by*, *bh*, *bw*]采用 L2 范数损失。特别注意，当 *po*=1 时，总损失是 3 个损失的加权和；当 *po*=0 时，由于没有对象，损失中只有 *po* 的损失才有意义，总损失就是 *po* 损失。可见，目标定位实质是分类（类别）和回归（边界框）任务之和。

预测新样本时，*po* 越接近 1 说明有对象的概率越高，接近 0 说明没有对象。[*c*1, *c*2, *c*3]最大位置给出对象类别，例如 *c*2 最大，则对象是摩托车；*c*1 最大，则对象是汽车等。最大值与其他值相差越多，属于该类别的概率就越高，概率 *pc* 由 softmax 函数给出。所以实践中，采用阈值法判断图像中是否有对象，如果 *po* 和 *pc* 乘积大于阈值，则存在对象，小于阈值则没有对象，阈值可以取 0.6，也可以取 0.7。有对象时[*bx*, *by*, *bh*, *bw*]是对象边界框向量。

目标定位的网络结构和分类网络完全一样，都是多层卷积层加全连接层，只是最后全连接层的输出向量不仅包含 *C* 个（类别数目）分

值向量，还需加上 5 个元素[*po, bx, by, bh, bw*]。

12.2　目标检测



图 12.2 目标检测

目标定位任务中图像只有一个对象，但目标检测任务中一般有多个对象，有时对象多达几十个。目标检测任务的核心思想就是多任务学习，即把每个对象的检测任务看成一个目标定位任务，同时完成多个目标定位任务。但是难度大很多，因为目标定位任务中对象比较大，而目标检测任务中，对象大小变化很大，小的对象可能只占整幅图像的 2%，大的对象可能占整幅图像的 50%，尺度差别有 20 倍以上。卷积网络不具有尺度不变性，很难同时处理好尺度差别这么大的对象，所以目标检测任务困难很多。

假设有 3 类对象，每个对象的目标定位任务需输出 ***ob***=[*po*, *c*1, *c*2, *c*3, *b*x, *b*y, *b*h, *b*w]向量，多个对象则需输出多个向量 ***ob***。这里存在一个问题，图像中对象数目不定，输出几个向量 ob 合适呢？解决方案是，根据先验知识，道路环境下对象数量一般在几十以内，则可以输出 49 个向量 ***ob***。这样绝大部分情况下都能适应，只有当对象数量超过 49 后，超过部分不能检测。如果其他应用，对象数量很少，当然也可以只输出 9 个向量 ***ob***。

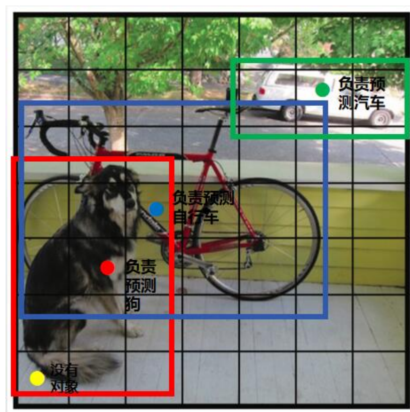


图 12.3 7×7 网格

现在假设图像有 3 个对象，如何制作该图像的标签呢？此时假设网络输出 49 个向量 ob，但图像只有 3 个对象，这 3 个对象如何分配到这 49 个向量中的 3 个呢？采用位置匹配法。具体方式如下：首先把整幅图像分割成 7×7 网格，因为输出有 49 个向量 ***ob***（7×7=49），然后根据每个对象边界框的中心位置落在哪个网格，则匹配给对应向量。因为图像分割为网格，所以输出的 49 个向量也按网格排列成 7×7。

图 12.3 中第 1 个对象汽车边界框的中心位置落在第 2 行第 6 列网格内，则匹配到对应的第 2 行第 6 列的向量。第 2 个对象自行车边界框的中心位置落在第 4 行第 3 列网格内，则匹配到对应的第 4 行第 3 列的向量。一定要注意，按对象边界框的中心位置进行一一匹配，按照这个原则，可以把所有对象匹配到相应的向量。如果某个网格内没有对象，即没有对象边界框的中心落入这个网格（如图 12.3 中第 6 行第 1 列网格），则对应位置向量的 *po*=0，其他 7 个元素毫无意义，不需指定。再次强调，某个网格没有匹配对象，不表明该网格内不存在任何对象，只是对象边界框的中心位置没有落入而已，边界框的其他部分可以落入该网格。所以一个对象可以覆盖多个网格，当然小对象有可能只位于 1 个网格内；一个网格可以被多个对象覆盖，也可以没有被对象覆盖或只被 1 个对象覆盖。这种方法的思想是向量只负责其对应位置处的对象检测，但对象的形状和大小不定。图像一般分割为 $n \times n$ 网格且 n 为奇数，通常取 3 到 13。因为大对象边界框的中心位置一般位于图像中心，如果 n 取偶数，如 2×2，则大对象无法匹配。

损失函数取所有向量 ***ob*** 的损失的和。在目标检测任务中，大部分网格没有和对象匹配，所以向量 ***ob*** 损失中 *po* 损失出现类别不平衡的问题，*po*=0 的样本远大于 *po*=1 的样本，可以采用第 11 章介绍的 FL 损失。向量 ***ob*** 损失中的类别损失和目标定位任务一样，采用 softmax 损失，边界框[*bx*, *by*, *bh*, *bw*]损失采用 L2 损失。

预测新样本和目标定位一样，如果某个向量 ***ob*** 的 *po* 和 *pc* 乘积大于阈值，则说明该向量预测到 1 个对象，对象位置由[*bx*, *by*, *bh*, *bw*]决定。

目标检测的网络结构和目标定位的网络结构不一样，目标检测的网络输出是 $7\times7\times8$ 的 3D 特征图，目标定位的网络输出是 8 维向量，可以看作 $1\times1\times8$ 的 3D 特征图，是目标检测的一种特殊情况，即图像分割为 1×1 的网格，相当于没有分割。网络结构上，目标检测网络全部由卷积层组成，不能加全连接层，因为全连接层容易丢失位置信息，导致对象的位置不准。

采用这种端到端的检测方法，读者容易犯的一个错误是，认为网络只是利用了对象所占据的局部区域信息进行预测，其他区域信息没有用来预测该对象。其实，网络利用的是整幅图像信息进行对象预测，即使对象很小只占据了 1 个网格，因为网络输出的 3D 特征图的每个深度维信息（就是一个向量 $\boldsymbol{ob}$）的感受野是整幅图像。由于有全局视野，这种方法不容易把背景区域误识别为对象，同时泛化性能好，由自然图像训练的检测网络，对非自然图像物体的检测效果好，能用于绘画作品中的物体检测。采用端到端学习方法，网络结构简单，检测速度较快。

采用全局视野，也有弊端，物体位置精准性差，边界框不准，同时难以检测出小物体，小物体的查全率较低。为了提高对小物体的检测效果，有 3 个手段，第一是增大输入图像尺寸，从 208×208 增加到 416×416，第二是增大网络输出的空间分辨率，比如从 7×7 增加到 13×13，第三是融合多尺度的特征图，获取不同分辨率，详见 12.6 节。

12.3　非极大值抑制 NMS

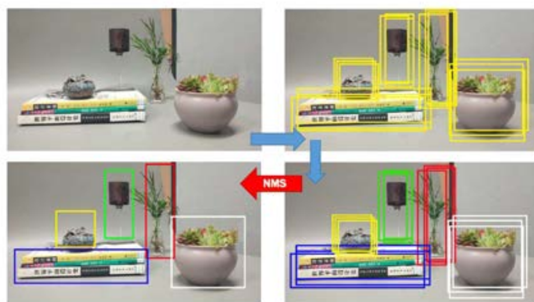


图 12.4 非极大值抑制

网络预测新样本时，每个网格输出 1 个边界框，预测 1 个对象。实际中经常会出现一个对象被相邻的多个网格预测，导致一个对象画出多个边界框。多个边界框中有很多位置不准，偏移比较大，需要综合这些边界框得到一个最准确的边界框，并去除多余的边界框。最简单常用的方法是非极大值抑制（Non-Max Suppression，NMS），即保留置信度最高的预测边界框，去除剩下的与该边界框类别相同且重合度比较高的其他边界框。

下面举例具体说明非极大值抑制算法。假设网络输出 7×7 边界框，那么对所有 49 个边界框计算置信度 $p=po \times pc$。首先去除置信度小于阈值（如 0.6）的边界框，置信度小于阈值说明存在对象的可能性很低，基本是背景。然后对剩下的边界框，取置信度最大的作为对象预测输出，表明检测到一个对象。接着去除所有与该边界框类别相同

且重合度大于阈值（如 0.5）的边界框。如果还有剩下的边界框则继续上面操作，取最大置信度的边界框作为对象预测输出，去除其他类别相同的相交边界框，直到没有边界框剩下。其核心思想很简单，以概率最高的预测作为对象“真身”输出，去除“真身”产生的其他概率更低的“假身”，然后依次检测其他对象。这里有一个核心假设是“假身”和“真身”是同一对象，且“假身”和“真身”重合度很高。如果重合度低于阈值，则认为是另一个对象产生的预测。该阈值对系统性能影响很大，如果取得大，则会留下大量“假身”；如果取得小，则会去除大量“真身”，特别是在对象重叠的时候。

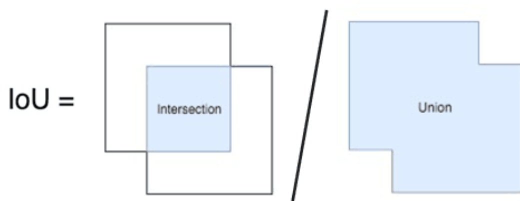


图 12.5 交并比（IoU）

衡量两个边界框的重合度采用交并比（Intersection over Union，IoU），即两个边界框的交集面积与并集面积之比，IoU 越大，说明两个边界框越重合。例如 IoU=1，表示两个边界框完全重合且形状一致，IoU=0 时，两个边界框完全不重合。

利用交并比可以衡量检测算法的性能，即当预测边界框与对象真实边界框的 IoU 大于阈值时，认为检测正确，阈值越大要求越严格，一般取 0.5。

12.4　锚点框 Anchor Box

上述检测算法有个缺点，一个网格只能检测一个对象，因为只输出了一个向量 ***ob***。现实中经常出现 2 个对象重叠的情况，它们边界框中心会落入同一网格，所以必须有手段处理这种情况，即要求每个网格能检测多个对象，输出多个向量 ***ob***，这些向量如何与对象匹配呢？采用形状匹配法。



图 12.6 每个网格有 3 个锚点框

具体方法是采用锚点框（Anchor Box）技术。假设每个网格输出 3 个向量 ***ob***，每个向量 ***ob*** 对应一个锚点框即水平矩形框，3 个锚点框的高宽比和大小各不相同，这些锚点框都位于网格内,形状和大小都是事先设计好的。

图像中如有 2 个对象落入同一网格，如何制作标签呢？计算对象边界框与 3 个锚点框的 IoU，如果最大 IoU 大于阈值（如 0.5），则对象与最大 IoU 的锚点框匹配，否则匹配失败。如果锚点框与某个对象

的 IoU 大于阈值，但没有与对象匹配，则忽略该锚点框的损失。剩下的锚点框向量 ob 中，元素 po 都等于 0。如图 12.6 所示，蓝色对象和白色对象都落入第 4 行第 5 列网格，每个网格有 3 个锚点框，一个正方形，两个矩形。水平锚点框与蓝色对象最匹配，竖直锚点框与白色对象最匹配，且它们 IoU 均大于阈值，故计算其损失。中间正方形锚点框与白色对象 IoU 大于 0.5，但没有与其匹配，所以忽略该锚点框损失。注意对象与某个锚点框最匹配，其 IoU 不一定大，有可能很低，对于这种情况，说明对象和锚点框形状大小差别很大，系统很难检测到这种对象。故对于与对象最匹配的锚点框，如果 IoU 小于阈值，该锚点框还是要作为负锚点框计算损失；只有大于阈值，该锚点框才能作为正锚点框计算损失。需要特别注意的是，计算锚点框与对象边界框的 IoU 时，锚点框要对齐边界框，即移动锚点框使其中心与边界框中心重合。这种方法的思想是锚点框只负责检测其对应位置处特定形状大小（锚点框的形状大小）的对象。

锚点框数量一般是 3 到 10 个，手工指定它们形状大小，尽可能与对象的形状大小一致，比如行人对象设计瘦高的锚点框，汽车尾部可设计正方形的锚点框，汽车侧面可设计相对细长的锚点框。还可采用 k 均值算法自动聚类出锚点框形状，即对所有对象的边界框进行聚类，取聚类中心作为锚点框，两个边界框的距离度量需采用 1-IoU。

锚点框技术大大增加了输出向量 ***ob*** 的数量，能检测到更多对象，特别是密集的部分重叠的对象，因此显著提高了查全率。当对象与锚点框的 IoU 接近 1 时，对象的检出率高且边界框精度高，但 IoU 比较小时，对象难以检测出且边界框不准确，所以锚点框技术难以应付对

象形状大小差别很大的情况。

由于卷积网络只能输出 3D 特征图，所以这些向量按固定顺序拼接成一个超级向量，网络输出为 7×7×24（24=8×3）。预测新样本时，也是采用非极大值抑制，流程不变。

虽然锚点框技术能处理多个对象落入同一网格，但当对象数量大于锚点框数量时，算法不能处理；或者 2 个对象与同一个锚点框匹配，也处理不好。不过这种情况很少，对性能影响不大，但需要有手段处理这些情况。

12.5　边界框参数

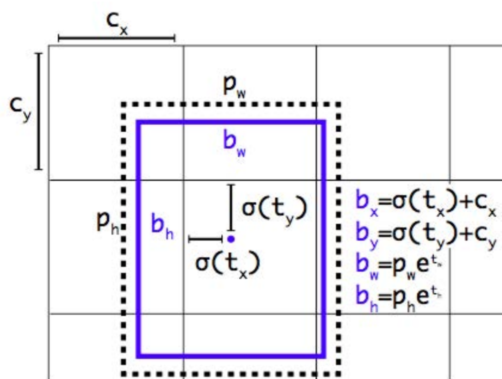


图 12.7 边界框参数（*tx, ty, th, tw*）计算

边界框参数（*bx, by, bh, bw*）如果按照第 12.1 节介绍的方法计算，

虽然都在 0 到 1 范围内，但变化会很大。比如对于大对象 bh 可能是 0.8，小对象可能是 0.1，这会导致平方损失函数对小对象预测不准确。以 bh 为例，因为平方损失为$(bh \quad bh_hat)^2$，其中 bh 是标签值，bh_hat 是预测值。当 bh 为 0.8 时，bh_hat 为 0.7，则损失为 0.01，预测相对误差为 0.1/0.8；当 bh 为 0.1 时，如果预测相对误差保持不变，则预测可为 0.2，但此时相对误差为 0.1/0.1，十分巨大。为了克服这个缺点，采用 sigmoid 和指数函数让参数接近 0。

首先建立图像坐标系，网格边长为单位长度。训练样本中，对象边界框中心位于第 2 行第 2 列的网格内,则位置基本偏移 Cx=1,Cy=1，对象中心相对于该网格左上角的偏移量为 $\sigma(tx)$和 $\sigma(ty)$，其中 σ 是 sigmoid 函数，因为 bx 和 by 是已知的，所以可求得 tx 和 ty。因为边界框中心一般位于网格中心，故 $\sigma(tx)$和 $\sigma(ty)$约等于 0.5，所以 tx 和 ty 约为 0。同理对于 bh 和 bw，根据匹配的锚点框尺寸 ph 和 pw，可计算出 th 和 tw。因为边界框与匹配的锚点框 IoU 比较大，故尺寸相当，所以 th 和 tw 约为 0。这样边界框参数由（bx, by, bh, bw）转化为（tx, ty, th, tw），且新参数约等于 0，与对象大小位置无关，有利于学习。新参数（tx, ty, th, tw）如果还采用 L2 损失，由于部分对象位置可能偏离网格中心较大，导致 tx，ty 绝对值很大；部分对象形状可能与锚点框的形状差别很大，导致 th，tw 绝对值很大。大参数（tx, ty, th, tw）会导致 L2 损失很大，从而很难训练。可以采用平滑 L1 范数损失，当参数很大时，损失为线性增长，增长比较缓慢；而 L2 范数是二次增长，增长很快。平滑 L1 范数损失公式如下：

$$smoothL1(x)=\begin{cases}0.5x^2 & if\ |x|<1\\ |x|-0.5 & otherwise\end{cases} \quad (12.1)$$

预测新样本时，网络输出向量 ***ob*** 输出边界框参数（*tx, ty, th, tw*），根据向量 ***ob*** 的位置，可得到基本偏移 *Cx* 和 *Cy* 的值；根据向量 ***ob*** 匹配的锚点框尺寸，可得到 *ph* 和 *pw* 的值；然后根据公式可得到（*bx, by, bh, bw*）参数。

12.6　特征金字塔网络 FPN

卷积网络有个特点，就是网络顶层提取的特征富有语义信息，底层提取的特征位置信息丰富。如果检测网络只输出顶层信息，则对象位置不准。能不能结合底层位置信息，来提高位置准确度？如果只是简单地输出底层信息，由于缺乏语义信息，容易导致输出虚假对象。需要结合顶层的语义信息和底层的位置信息来提高检测效果，FPN（Feature Pyramid Networks）特征金字塔网络就是对这种思想的一种很好实现。

基础网络是训练 ImageNet 得到的预训练模型（如 ResNet）去掉顶层的全连接层和平均池化层。基础网络一般都是分阶段的，每个阶段都是连续几个卷积层，这些卷积层的空间分辨率大小一致，一般后一阶段的分辨率是前一阶段的一半。阶段之间采用步长为 2 的池化层或卷积层进行下采样。取每个阶段最后的卷积层作为 FPN 的特征层，对这些特征层进行融合，利用融合后的特征进行对象检测。

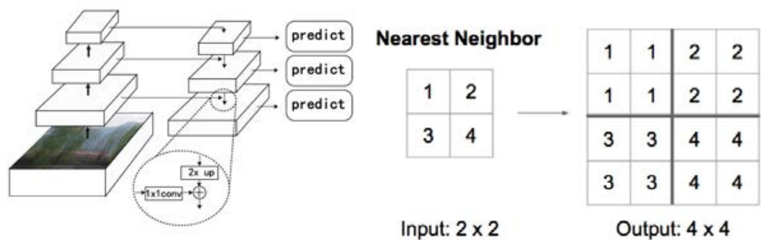


图 12.8 FPN 特征金字塔网络及最近邻上采样

假设融合 3 个尺度的特征，基础网络最后 3 个阶段的最后的卷积层分别为 13×13×1024，26×26×512，52×52×256。每个网格输出 B=3 个锚点框，有 C=20 类对象。

首先对顶层 13×13×1024 特征图进行 1×1 卷积，得到 13×13×256 特征图 s1，然后预测子网络进行预测。这样完成了该尺度的预测，主要用于检测大对象。

然后对中间层 26×26×512 特征图进行 1×1 卷积，得到 26×26×256 特征图 s2_，对特征图 s1 进行上采样得到 26×26×256 特征图 s1_，s1_和 s2_两个特征图进行逐元素相加，得到特征图 s2，然后预测子网络进行预测。这样就完成了该尺度的预测，主要用于检测中等对象。

同理，对低层 52×52×256 特征图进行 1×1 卷积，得到 52×52×256 特征图 s3_，对特征图 s2 进行上采样得到 52×52×256 特征图 s2_，s2_和 s3_两个特征图进行相加得到特征图 s3，然后预测子网络进行预测。这样就完成了该尺度的预测，主要用于检测小对象。

这里面有 3 种路径，基础网络从 52×52×256 层到 13×13×1024 为从下至上路径（Bottom-up），用于提取特征；FPN 从 13×13×256 到 52×52×256 为从上至下路径（Top-down），用于把顶层语义信息引入底层；基础网络到 FPN 即 1×1 卷积为横向路径（Skip-connect），用于特征抽象。横向路径必须有卷积，不能直连，FPN 采用 1×1 卷积，也可以采用 3×3 卷积。

其中特征图 s1，s2 和 s3 的深度必须相等（常用 256）。对特征图进行上采样可采用最简单的最近邻（Nearest Neighbor）方法，即把每个元素复制 4 份，也可以采用复杂的转置卷积（Transpose Convolution）。

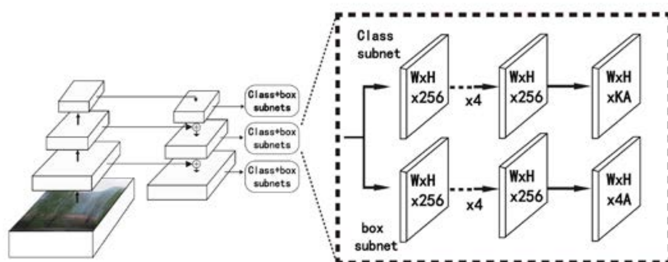


图 12.9 FPN 检测网络

3 个尺度的预测子网络结构相同，且权重共享，但类别和边界框采用 2 个网络分开检测。对融合得到的 $W \times H \times 256$ 特征图进行类别检测时，先进行 4 次连续 3×3 卷积，每个卷积后跟 ReLU 激活，不改变特征图的深度。最后进行 3×3 卷积，不用激活，输出 $W \times H \times [(1+C)B]$，预测是否存在对象和对象类别。

对融合得到的 $W \times H \times 256$ 特征图进行边界框检测时，也是先进行 4 次连续 3×3 卷积，每个卷积后跟 ReLU 激活，不改变特征图的深度。最后进行 3×3 卷积，不用激活，输出 $W \times H \times [4B]$，预测边界框参数。注意类别和边界框 2 个子网络虽然结构相同，但参数不同。采用 2 个网络是因为类别和边界框任务差别很大，用一套参数效果较差。

预测子网络也可以采用更简单的结构，如只有 1 个 3×3 卷积，以提高检测速度。甚至类别和边界框 2 个子网络合并为 1 个网络。

实践中一般融合 3 个尺度的特征，尺寸分别是输入尺寸的 1/8、1/16、1/32，为了进一步提高大对象检测性能，还可融合尺度为 1/64、1/128 的特征。FPN 能极大提高目标检测网络性能，几乎成为标配。

每个尺度采用 3 个锚点框共 9 个锚点框，对于 COCO 数据集，这些锚点框为(10 × 13)，(16 × 30)，(33 × 23)；(30 × 61)，(62 × 45)，(59 × 119)；(116 × 90)，(156 × 198)，(373 × 326)。

12.7 YOLO 算法

把上面的所有组件组合就是 YOLO（You Only Look Once）算法。首先把图像分割成 $S \times S$ 网格，每个网格输出 B 个锚点框，对象有 C 个类别，则网络输出为 $S \times S \times [(1+4+C)B]$，网络结构采用全卷积层，FPN 网络融合 3 个尺度的特征图提高小对象的检测性能。

制作训练集样本时，根据对象边界框中心位置和形状，找到匹配的锚点框并进行参数转化，忽略与对象 IoU 较大的但未匹配的锚点框，

剩下的大量未匹配的锚点框 po=0，其他 4+C 个值不需指定。损失函数是所有向量 ***ob*** 损失之和，每个向量 ***ob*** 的损失是 3 部分损失的加权和。预测新样本时，先去除置信度低的锚点框，然后进行非极大值抑制获得预测对象。

一些典型超参数如下：输入图像尺寸为 416×416 或 208×208，S 取 7 或 13，B 取 3 或 5。

因为目标检测的数据集不够大，所以采用迁移学习进行网络训练。对于 YOLO v2 先在 ImageNet 上训练分类网络，分类网络是 Darknet-19，采用平均池化层输出 1000 维分值向量。进行多尺度训练，先进行 224×224 尺寸 160 轮（epoch）训练，然后进行 448×448 尺度训练 10 轮。输入 448×448 时，DarkNet-19 达到了 top-1 准确率 76.5%，top-5 准确率 93.3%。DarkNet-19 由 19 个卷积层和 5 个池化层组成，类似 VGG 结构，大量采用 1×1 卷积，引入 BN 用于稳定训练，加快收敛，同时防止模型过拟合。

训练检测网络时，输入是 416×416，输出是 13×13×125。模型去掉分类网络的平均池化层及最后 1 个卷积层，取而代之的是 3 个 3×3×1024 的卷积层，并且每个新增的卷积层后面接 1×1×125 的卷积层，训练 160 轮。输入为 416×416 时，模型在 COCO 数据集获得了 44.0% 的 mAP-50 性能，在 Geforce GTX Titan X 检测速率达到 67FPS。

YOLO v3 模型是 DarkNet-53，由 53 个卷积层组成，没有池化层，引入 ResNet 结构。采用 FPN 结构极大提高了性能。输入为 416×416 时，模型在 COCO 数据集获得了 55.3% 的 mAP-50 性能，在 Geforce GTX

Titan X 检测速度是 29ms/张。

上面介绍的目标检测方法是端到端学习方法，称为一阶段检测方法。还有一种所谓二阶段检测方法，以 R-CNN 开端，以 Faster R-CNN 和 Mask R-CNN 为代表，该类方法需要先产生候选区域，再进行分类，而 YOLO 方法不需要产生候选区域，直接得到结果。曾经普遍认为，二阶段检测方法精度高，速度慢，一阶段检测方法精度低，速度快。但目前一些一阶段检测方法精度达到甚至超过二阶段检测方法，二阶段检测方法速度也变得很快，两种方法之间差别越来越小。但长久来看，可能一阶段检测方法更有前途。

虽然目标检测算法取得了不错性能，但与人类水平相比，差距很大，而分类任务已经接近人类水平了，这说明目标检测任务远难于分类任务。这是因为目标分类任务中，图像只有一个大对象且对象位置基本居中，而检测任务中，对象的数目、大小和位置均不确定，而且差别很大，加上对象有重叠遮挡，针对这些难点，发展了很多技术，下面介绍一些代表技术。

12.8　软非极大值抑制 Soft-NMS

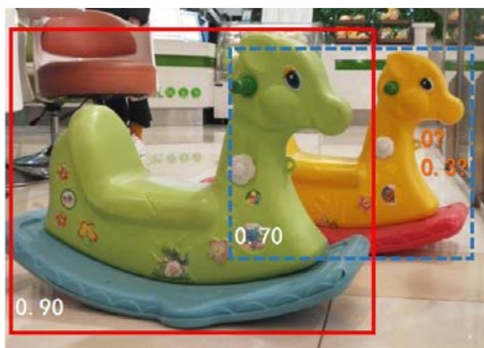


图 12.10 两个重叠边界框

NMS 方法中，如果两个边界框的 IoU 大于阈值，则置信度低的边界框被认为是虚假对象，直接被抛弃。但实际中经常出现两个对象高度重叠 IoU 大于阈值，直接抛弃低置信度对象会造成大量的漏检，如图 12.10 中两匹马，会漏检置信度为 0.7 的马，只检测出置信度 0.9 的马。Soft-NMS 方法不直接抛弃置信度低的边界框，而是降低其置信度，使之有机会再次被检测为对象。置信度被降低的程度与其与最大置信度边界框的 IoU 相关，IoU 越大说明重叠越大，越有可能是同一对象，故降低程度应该越大。可以采用高斯函数来表达：

$$s = s \cdot e^{-iou(M,b)^2/\sigma} \quad (12.2)$$

其中 s 是边界框 b 的置信度，σ 是参数，常取 0.5，M 是最大置信度边界框。当 IoU 为 0 时，s 不变，当 IoU 为 0.5 时，s 变为原来的 0.6，

当 IoU 为 1 时，*s* 变为原来的 0.14。这样没有重叠的边界框，置信度不受影响，重叠越大的边界框受影响越大。

Soft-NMS 算法流程和 NMS 几乎一样。首先去除置信度小于阈值（如 0.5）的边界框，然后从剩下的边界框中取置信度最大且大于阈值的边界框 M 作为对象预测输出，表明检测到一个对象，接着采用高斯函数来更新剩下所有边界框的置信度。对更新后的边界框继续同样操作（即取置信度最大且大于阈值的边界框 M 作为对象预测输出，接着采用高斯函数来更新剩下所有边界框的置信度），直到剩下所有边界框的最大置信度小于阈值，表明不再有对象存在，结束流程。

Soft-NMS 能较大地提高检测算法性能（特别是预选边界框很多的情况下），同时不改变整个检测算法框架，不需重新训练网络，能很方便地植入到现有检测算法中，且几乎不影响检测速度，因此被广泛采用。

12.9 聚焦损失 Focal Loss

第 11 章讲述类别不平衡时已经简单介绍了 FL（Focal Loss），由于 FL 简单明了，效果很好，广泛应用于类别不平衡和检测算法中，因此本节详细介绍它。类别不平衡时，模型损失会被大类所主导，导致大类很快就收敛，损失很低，但小类不能有效学习，损失很大，最终模型不能对小类进行正确分类。如果能降低大类损失，让模型损失聚焦于小类，则能提高小类学习效果。这里的关键是，大类学习快，概率很快接近 1，如果能减小概率接近 1 的损失，则就能抑制大类。

所以根据这点，提出了 FL，公式如下：

$$FL(p_t) = -(1-p_t)^{\gamma}\log(p_t) \quad (12.3)$$

其中 p_t 是样本的概率，γ 是大于 0 的参数，通常取 2，γ 为 0 时就是传统损失。

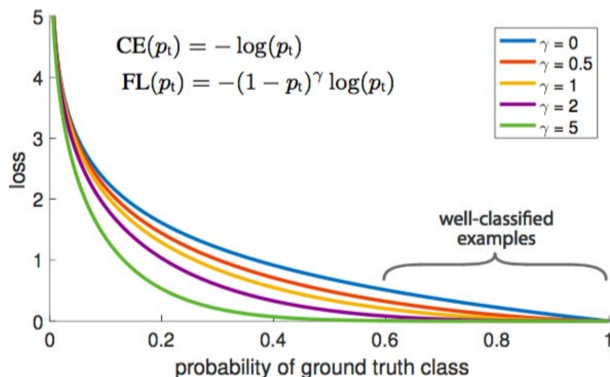


图 12.11 FL 图像

当概率接近 0.6 时，表明样本被正确分类，FL 相对传统损失，被严重挤压，变得很小；概率接近 0.2 时，表明样本错误分类，FL 相对传统损失，挤压轻微，变化不大。故 FL 能自动聚焦于难样本，让网络重点学习难样本。

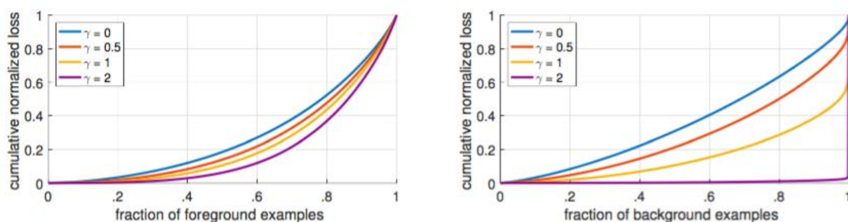


图 12.12 FL 聚焦于难样本

目标检测网络中，大量的锚点框没有对象，只有极少的锚点框有对象，所以是典型的类别不平衡问题。传统的方法采用在线难样本挖掘技术（Online Hard Example Mining，OHEM)来聚焦于难样本，但是这种方法完全抛弃易样本，而 FL 只是降低易样本的损失，因此 FL 效果更好。实验结果如图 12.12 所示，显示的是累计样本损失，横坐标是按样本损失从小到大排序，纵坐标是累计损失。左边是对象损失图，FL 中 20%的样本（横坐标从 0.8 到 1）损失占总损失的 60%（纵坐标从 0.4 到 1），传统的也占到 50%左右，可见 FL 和传统损失图很接近。右边是背景损失图，FL 中不到 1%的难样本（横坐标 1 附近）的损失占总损失的 95%以上（纵坐标从 0 到 1），可见 FL 和传统损失图差别很大。FL 能自动聚焦于背景（大类）中的难样本，易样本的损失几乎为 0，对于对象（小类）影响很小。FL 只需简单修改损失函数，而且对参数 γ 不敏感，十分容易推广。

12.10　基础网络 Backbone

目标检测基础网络大多采用 ImageNet 分类预训练网络，如

ResNet，但分类任务和检测任务差别很大。分类任务不需要位置信息，可以对输入图像进行大比例的下采样，如常用的都是 32× 的下采样，把输入 224×224 的图像下采样为 7×7（224/7=32）。大的下采样比例会丢失对象位置信息，导致大对象的边界框不确、小对象可能检测不到，影响性能，所以检测网络的下采样比例不能过大，可以采用 16×。分类网络一般分 5 个阶段，每个阶段由多个连续卷积组成，阶段之间由步长为 2 的池化层或卷积层连接，进行 2× 的下采样。检测网络为了能更好的检测大对象，需要更多的阶段，如 6 个阶段。根据这些原则，设计了 DetNet，网络共 6 个阶段（第 1 个阶段 2× 没有画出），最后 2 个阶段不进行下采样，保持 16× 的尺度，有利于网络保留位置信息。由于最后 2 个阶段没有下采样，如果采用传统 3×3 卷积，网络的计算量会增加，神经元感受野会减小。采样扩张卷积（Dilated Conv）可以在不增加计算量的同时保持感受野不变小。传统 3×3 卷积核元素是相邻的，感受野和卷积核尺寸一致；而扩张卷积元素有间隔，感受野大于卷积核尺寸。最常见的是 3×3 扩张卷积，元素间隔为 1，这样其等效感受野为 7×7，相当于 3 个连续的 3×3 卷积核的感受野，但参数量和计算量只与 1 个 3×3 卷积核相同。

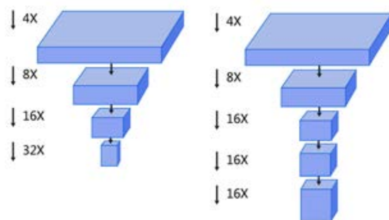


图 12.13 分类和检测的基础网络

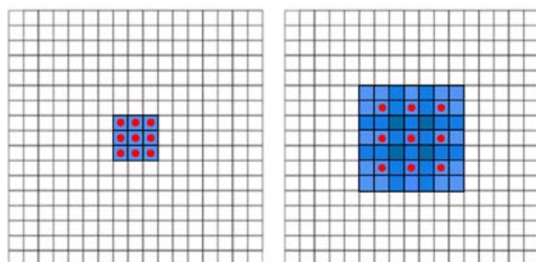


图 12.14 3×3 扩张卷积

DetNet 网络的实现采用了 ResNet 网络的瓶颈（Bottleneck）结构，来减小计算量和增加网络深度。Bottleneck 结构的核心思想是大量采用计算高效的 1×1 卷积核对特征图进行深度维特征变换，轻量网络中也是采用相同技术。DetNet 网络前面阶段和 ResNet 一样，只是最后 3 个 16×尺度阶段都输出 256 个特征图，把 3×3 卷积变为扩张 3×3 卷积，并采用 FPN 进行特征综合来提高性能，由于特征图空间分辨率相同，故不需要进行上采样，直接连接就可以。有一个细节需要注意，ResNet 网络中不同阶段的尺度不同是通过下采样实现的，但 DetNet 的最后 3 个阶段尺度相同，卷积层都由相同的扩张卷积的瓶颈连接，并没有明显的阶段界线。DetNet 采用 1×1 投影（Projection）卷积来进行阶段划分，能明显提高模型性能。

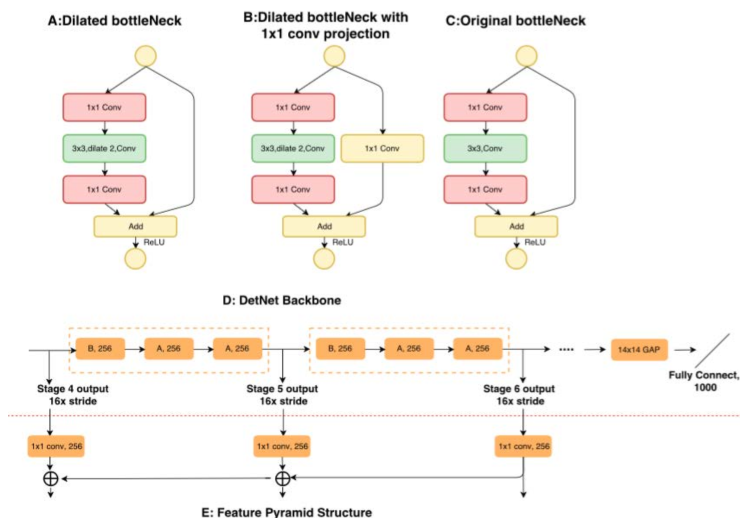


图 12.15 DetNet 结构

12.11　多尺度 Multiple Scales

由于卷积网络不具有尺度不变性，所以检测网络难以处理对象尺度差别很大的情况，上面介绍的 FPN 和 DetNet 本质上都是为了解决尺度问题，采用多层特征金字塔来同时检测尺度不一的对象。由于网络要同时处理多尺度的对象，可能会导致网络难以训练，效果有限。

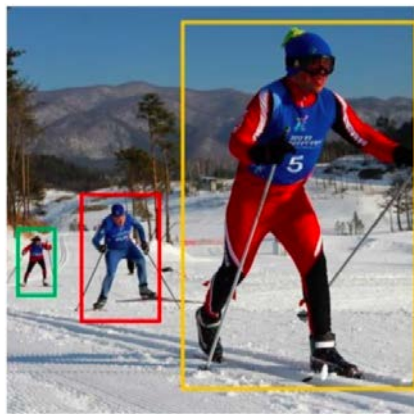


图 12.16 多尺度对象，3 个尺度差别很大的人

下面做 3 个实验来说明卷积网络不具有尺度不变性。实验 1 是，用尺度为 224×224 的图像作为输入来训练 ImageNet 分类器。测试时采用不同尺度的图像，这些图像都来自 ImageNet 数据集，先通过下采样来降低尺度，然后再通过上采样变成尺度为 224 的图像作为分类器的输入。结果表明尺度为 224×224 时，top-1 准确率为 76.8%；尺度为 128×128 时，top-1 准确率为 67.1%；尺度为 96×96 时，top-1 准确率为 59.7%。由此可见，尺度越小，效果越差，网络难以同时处理多尺度图像。

实验 2 是，用尺度为 96×96 的图像作为输入来训练 ImageNet 分类器，测试时也用尺度为 96×96 的图像，结果是 top-1 准确率为 69.2%，与第 1 个实验的结果 59.7%相比，差别很大。这表明要达到好效果，训练网络的图像尺度要和测试时的一致。

实验 3 是，首先用尺度为 224×224 的图像作为输入来训练

ImageNet 分类器，然后用尺度为 96×96 的图像上采样到 224×224，对模型进行微调，最后测试时用尺度为 96×96 的图像上采样到 224×224，结果是 top-1 准确率为 72.1%。与 59.7%相比，这表明微调网络可以使网络适应新尺度。

SNIP（Scale Normalization for Image Pyramids）根据这些特点，设计了尺度归一化网络，核心思想是：网络只处理一种尺度的对象，通过对图像进行上采样检测小对象，进行下采样检测大对象（相当于实验 3 微调网络）。如图 12.16 所示，网络只检测中间的人。对于前面大尺度的人，缩小图像尺寸，使之变得和中间的人差不多大，然后检测。对于后面小尺度的人，放大图像尺寸，使之变得和中间的人差不多大，然后检测。这样，网络只需处理一种尺度，能达到最好性能。

具体实现细节如下，假设网络有 3 种尺度的输入(480，800)、(800，1200)和(1400，2000)，也就是把样本图像改变尺寸（rescale）到这 3 种尺度。训练时，对于每个尺度输入，网络只处理边界框大小位于一定范围内的对象，对象的边界框在范围之内的称为有效对象，在范围之外的称为无效对象。只有有效对象与锚点框进行匹配，计算损失，如果锚点框与无效对象的 IoU 大于 0.3，则忽略该锚点框的损失。预测时，3 个尺度独立进行预测，同样只有预测边界框位于范围内的才是有效的预测。然后 3 个尺度改变尺寸到同一尺度（预测边界框也跟着改变大小）融合为一个输出，最后进行 NMS。这样网络训练和预测都是处理同一尺度的对象，根据实验 1 和实验 2，能达到很好效果。SNIP 缺点是预测速度慢，因为要预测 3 个尺度的输入，才能得到最后的结果。

12.12　三叉戟网络 TridentNet

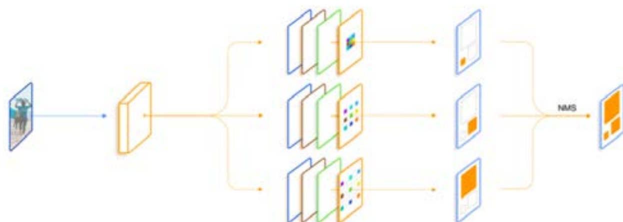


图 12.17 TridentNet 结构

尺度问题是目标检测的难点，FPN、DetNet 和 SNIP 是解决尺度问题的代表方法。FPN 采用一种尺度输入，利用多层特征金字塔来检测尺度不一的对象。SNIP 采用多尺度输入，每个尺度下只检测特定大小的目标。DetNet 利用扩张卷积实现小目标检测。TridentNet 综合了这些方法的优点，获得了极大的性能提升，同时没有增加计算量和权重数量，是一种非常实用的方法。

TridentNet 利用扩张卷积来实现不同尺度目标的检测。扩张卷积元素有间隔，常见的有 3×3 扩张卷积，元素间隔为 1 或 2，间隔为 0 就是传统卷积。间隔越大，有效感受野越大，越有利于大目标的检测。通过一个实验来说明，使用 ResNet50 作为基础网络，只改变最后一个阶段中每个 3×3 卷积的间隔，结果表明不同尺度目标的检测性能和间隔正相关，即有效感受野大，有利于检测大目标；有效感受野小，有利于检测小目标。

为了检测各种尺度目标，一个直接的方式就是使用多个并行分

支，每个分支的间隔不同。采用 3 个分支，间隔分别为 0、1 和 2 的网络结构称为三叉戟网络。具体细节为，当使用 ResNet50 作为基础网络时，去掉全连接层和第 5 阶段的卷积层，前 3 个阶段卷积层保持不变。第 4 阶段卷积层复制 3 份，作为 3 个并行分支，分别使用间隔为 0、1 和 2 的卷积。每个分支分别进行预测，预测结果合并后采用 NMS 作为最后输出。

3 个分支的权重共享可以进一步提高性能，原因有两方面，一是减少了权重数量以及潜在的过拟合风险，二是充分利用了每个样本，同样一套权重在不同间隔下训练了不同尺度的样本。最后还可以借鉴 SNIP，每个分支只训练一定尺度范围内的样本，避免极端尺度物体对性能产生影响。相邻分支之间的尺度可以有重叠，比如对于短边长度为 800 像素的输入图像，3 个分支样本面积的平方根范围分别为[0, 90]，[30, 160]，[90, ∞]。

TridentNet 在测试阶段需要运行 3 个分支，比起基础网络需要一些额外计算。为了不引入任何额外计算，在测试阶段，可以只保留间隔为 1 的分支来近似完整 TridentNet 的结果，性能下降很小。这里有两点要注意：第一，网络训练时仍必须采用 3 个分支；第二，所有样本在 3 个分支都训练，不能采用 SNIP 方法，因为最后如果只保留一支，那么权重最好在所有样本上的所有尺度上都充分训练。单分支预测方法的性能比 FPN 要更好。

12.13　人脸关键点定位



图 12.18 人脸关键点，分别为 5、49、68、194 个关键点

人脸关键点定位任务（face landmark）是定位出眼睛眉毛鼻子嘴巴和轮廓，最常用的是 5、49、68 或 194 个关键点，5 个关键点只能给出粗略位置，49 个关键点能描绘轮廓，68 个关键点描绘脸部外轮廓，194 个关键点可以近似认为是线模型。人脸关键点定位任务和目标检测任务一样也是位置敏感性任务，但难度低很多，因为人脸可以近似认为是刚体，变形小，五官位置大小基本固定，但头部姿态和表情会影响关键点位置。人脸关键点定位是很多人脸分析任务的基础，如人脸验证，表情分析和人脸识别。

人脸关键点定位任务先用人脸检测算法定位出人脸的边界框，只需在边界框内定位关键点，边界框尺寸需统一缩放至固定尺寸（如 60 × 60 或 40 × 40），定位网络一次只处理一张人脸。人脸检测算法可以采用目标检测算法如 YOLO，由于只需检测人脸，故基础网络可以设计得很轻量。

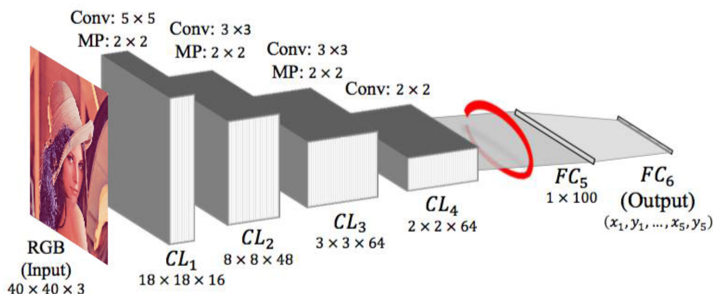


图 12.19 定位卷积网络

假设 m 个关键点坐标为 $\boldsymbol{P}=(p1,\cdots,pm)=(x1,y1,\cdots,xm,ym)$，在边界框坐标系进行归一化，这样卷积网络只需对这 $2m$ 个实数进行回归，可以通过全连接层实现。如图 12.19 所示，通过 4 个卷积层和池化层把输入 40×40×3 转化为 2×2×64，最后通过 2 个全连接层输出 10 个实数来定位 5 个关键点。损失函数可采用 L2 范数，也可采用加权损失，即用两眼瞳孔距离进行加权，权重为瞳孔距离平方的倒数。这种方法虽很简单，但效果不错。

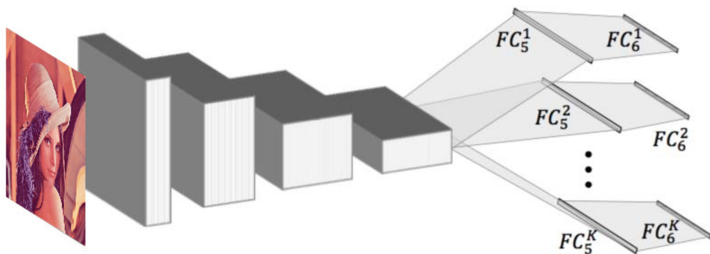


图 12.20 微调卷积网络 TCNN

为了进一步提高性能，比如大侧脸或夸张表情时定位精度，需要考虑头部姿态和表情的影响。为此发展了很多方法，比如辅助任务法、迭代校正法，这里介绍一种简单的微调法，该方法本质是对人脸进行无监督聚类，然后对每个聚类采用不同的权重进行训练。每个聚类代表不同的头部姿态或表情，这样网络就能自适应的学习，获得更好的效果。

该方法巧妙处在于不是采用图像的 RGB 像素进行聚类，而是采用预训练的卷积网络（如图 12.19 所示）的倒数第 2 个全连接层 FC_5 特征进行聚类。训练的具体过程为，先训练图 12.19 的网络，然后训练样本根据 FC_5 特征进行聚类，最后采用 FC_5 特征作为输入，对每个聚类分别训练一个线性回归器。预测时，先计算人脸的 FC_5 特征并判断其聚类类别，然后采用对应的线性回归器进行定位。聚类算法采用混合高斯模型，聚类数 K=64。训练线性回归器时，由于每类样本数很少，为避免过拟合，采用特殊的数据增强技术来扩增样本。

12.14 单个人体关键点定位



图 12.21 人体关键点

人体关键点定位任务是定位出头部脖子肩膀手臂腿等关节位置，是行为分析、人机接口和人体动画的基础。表面上看，人体关键点定位和人脸关键点定位一样，但人体四肢变形很大，可以认为是变形体，关键点位置千差万别，难度大很多。人体关节位置不管怎么变化，总受到人体结构的约束，人体总姿态的信息能提高关节定位精度。由于关节被遮挡、服饰的掩盖或附近其他人关节的干扰，只通过关节局部区域难以精确定位关节，而人脸关键点基本可以通过局部区域判断，这是两者的本质差别。要获得人体总姿态，需要全局视野，这要求网络融合多尺度信息，所以网络与前面介绍的 FPN 网络高度相似。

具体来说，人体关键点定位任务采用沙漏模块（Hourglass），沙漏模块包括 3 条路径 Bottom-up 路径、Top-down 路径和 Skip-connect 路径，这与 FPN 网络一致，路径连接方式也和 FPN 网络一致。与 FPN 网络不同的是，沙漏模块高度对称，Bottom-up 路径和 Top-down 路径完全对称，而 FPN 的 Bottom-up 路径厚重，Top-down 和 Skip-connect 路径轻量。图 12.22 中每个方框代表相同的卷积层，输出 256 个特征图。卷积层可以采用 ResNet 模块，即采用 1×1 卷积降维输出 128 个特征图，3×3 卷积输出 128 个特征图，再 1×1 卷积升维输出 256 个特征图。沙漏模块通过 4 次下采样，把输入 64×64 缩小为 4×4，再通过 4 次上采样，把 4×4 放大至原始输入尺寸，沙漏模块在最后的原始尺度输出预测结果。

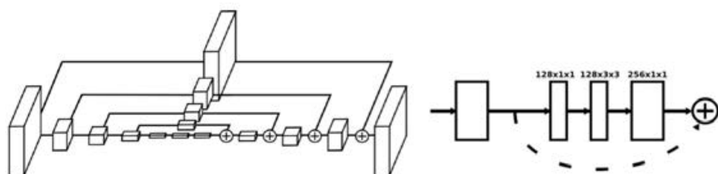


图 12.22 沙漏模块（Hourglass）和 ResNet 模块

整个定位网络为，输入为 256×256 包含人体的图像，经过 1 个步长为 2 的 7×7 卷积、1 个 ResNet 模块和 1 个步长为 2 的最大值池化层，尺寸缩小为 64×64，然后通过沙漏模块，最后接 2 个步长为 1 的 1×1 卷积，得到最终输出。网络最终输出并不是关键点坐标值，而是热点图。假设定位 15 个人体关键点（四肢各 3 个，躯干 2 个，头 1 个），则热点图是 64×64×15 的特征图，其含义是每个 64×64 特征图表示 1 个关键点的预测结果，特征图中每个位置的值表示关键点概率，故最大值位置为关键点预测位置。所以网络的损失函数就是预测热点图与真实热点图的 L2 范数，真实热点图是以关键点位置为中心的高斯分布，即关键点位置概率为 1，以高斯曲线向四周趋近 0，当方差为 1 个像素时，3 个像素之外的概率接近 0。



图 12.23 网络输出的热点图（Heatmap）

为了提高网络性能，可以堆叠多个（2 个到 8 个）沙漏模块，充

分挖掘多尺度信息。由于网络加深，容易出现梯度消失，沙漏模块为此采用特殊的连接方式，使每个沙漏模块产生监督信号，进行梯度反传。沙漏模块向后传输时，分成两路，上路接后面的沙漏模块，下路输出热点图（小方框），热点图计算损失进行梯度反传。再对热点图进行 1×1 卷积，使输出通道数和中间层相同，然后两路相加，注意还要加上前面沙漏模块的输出（虚线所示）。如果只采用上路连接，则没有中间监督信号，网络难以训练；如果只采用下路连接，则由于热点图的通道数（15）远小于其他卷积层的通道数（256），造成信息瓶颈导致信息丢失，网络性能下降。虽然沙漏模块结构相同，但是权重独立，各不相同。这种连接方式，相当于后面的沙漏模块对前面模块输出的热点图进行修正，是一种迭代处理方式，越到后期，增益越不明显，沙漏模块 4 个以上时，性能提升不明显。预测时采用最后一个沙漏模块的热点图。

这种方法同样能用于各种关键点定位任务，如人脸关键点定位。但这种方法只能对单个人体进行关键点定位，所以需要先检测人体边界框。

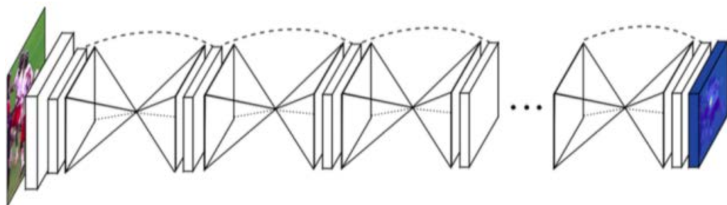


图 12.24 堆叠多个沙漏模块构成定位网络

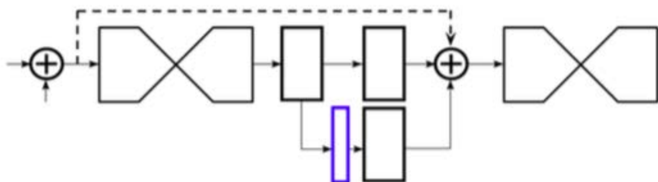


图 12.25 沙漏模块连接方式

12.15 多人人体关键点定位

多人人体关键点定位任务有两种方法，一种是从上至下（Top-down）方法，先进行人体检测，然后对每个人体进行单个人体关键点定位。人体检测算法可以采用 YOLO 算法，单个人体关键点定位可以采用沙漏算法。这种方法的最大缺点是检测速度与人体数量成正比，人越多速度越慢，在人体密集的场景难以推广。同时关键点定位精度受人体检测算法影响较大，人体边界框的轻微偏移有时会对关键点位置产生很大影响，特别地，在人体被遮挡或重叠时，很难获得精确的边界框。另一种是从下至上（Bottom-up）方法，这种方法不需要进行人体检测，而是先同时定位所有人体关键点，然后对关键点进行匹配连接，形成单个人体。这种方法的最大优点是检测速度基本不受人体数量的影响，能应用于各种场景，包括人体密集的场景。最大的挑战在于准确定位关键点，因为算法需要在不同尺度、遮挡或重叠等情况下精确定位关键点，关键点定位不准会导致关键点匹配错误。目前从上至下方法精度高，速度慢；从下至上方法精度低，速度快，

长期来看，从下至上方法更有优势，本节介绍这种方法。

从下至上方法有两个核心子任务：关键点定位和关键点匹配连接，都是多任务学习问题。关键点定位可以采用与单人人体关键点定位相同的方法，利用 CNN 网络强大的拟合能力，输出各关键点位置的热点图。关键点匹配连接需要正确连接同一个人体的各个关键点，形成完整人体。由于人体相邻关键点均由躯干连接，如肘关节和腕关节由小臂连接，颈关节和腰关节由上身连接等，要指引网络学习到这些知识，为此建立相邻关键点亲和场（Part Affinity Fields，PAF），PAF 建立相邻关键点之间的量化关系。这两个子任务既紧密联系又差别较大，紧密联系是指子任务都和关键点位置相关，差别较大是指关键点定位主要由关键点附近区域决定，关键点匹配连接主要由相邻关键点之间的区域决定。所以网络结构设计上，既要有相同的主干，又要有较明显的分支。

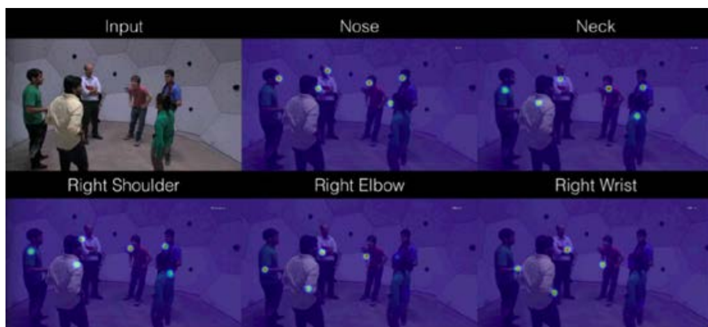


图 12.26 关键点热点图



图 12.27 相邻关键点（左肘左腕）之间的亲和场 PAF

具体来说，网络输入为 368×654×3 图像，经过主干网络为 VGG-19 的前 10 个卷积层（权重初始为预训练模型，然后进行微调），得到特征图 F，特征图 F 作为两个分支的输入。输入图像经过 3 个池化层，尺度缩小 8 倍，特征图 F 为 46×82×512。两个分支网络结构完全相同，都是 3 个连续的 3×3 卷积接 2 个 1×1 卷积，注意没有池化层。上层分支预测热点图，下层分支预测 PAF。分支网络和沙漏模块完全不同，分支网络没有对特征图进行 Bottom-up 和 Top-down 处理，尺度始终保持不变。这是因为这两个子任务主要由关键点局部区域决定，需要精确的位置信息，而不是特别需要全局信息，所以不能降低特征图空间分辨率。而单个人体关键点定位由于没有 PAF，网络不能提供关键点之间的连接关系，所以需要很强的全局信息，要降低特征图空间分辨率到很小的尺寸如 4×4。下面详细介绍两个分支的标签设计。

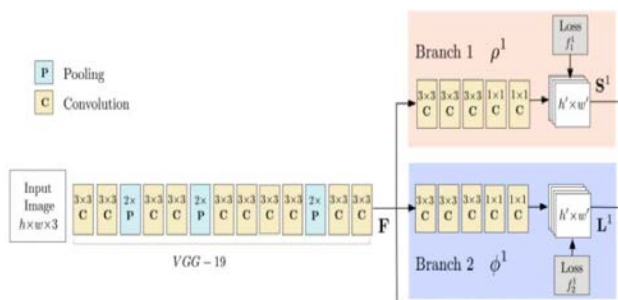


图 12.28 主干和两个分支构成的定位网络

上层分支是预测热点图，所以标签就是热点图，和单人人体关键点定位采用的热点图很类似，也是每种关键点一个热点图，即所有人的同种关键点都在一个热点图内，比如右肩热点图，右腕热点图。热点图内每个人关键点处形成一个热点，峰值为 1，以高斯曲线向四周趋近 0。热点图分支输出特征图的通道数等于关键点种类数。损失函数是预测热点图与真实热点图的 L2 范数。测试时，对热点图采用 NMS 算法，从而获得局部极大值作为关键点的位置。

下层分支是预测 PAF，所以标签就是 PAF。先说明两个概念。两个关键点相邻，是指它们之间由躯干直连，中间不能存在其他关键点，比如右肘关键点和右腕关键点是相邻关系，而右肩关键点和右腕关键点不是相邻关系，因为中间有右肘关键点。相邻关键点方向是由靠近躯干的关键点指向远端关键点比如右肘指向右腕，或上面关键点指向下面关键点比如颈部关键点指向腰部关键点。PAF 就是在相邻关键点之间建立单位矢量场，矢量方向为相邻关键点方向，其他位置处矢量为 0，这样量化了相邻关键点之间的关联。具体地，和关键点热点图

一样，也是每种相邻关键点建立一个 PAF，即所有人的同种相邻关键点都在一个 PAF 内，比如左肘左腕 PAF、左肩左肘 PAF 和颈部右腰 PAF 等。为了提供更多更强的学习信号，相邻关键点之间的单位矢量场并不只存在两个关键点的连线上，而是在一个矩形区域内都存在矢量场，即考虑了躯干的宽度，比如小臂的宽度。当多人小臂有重叠时，由于每个小臂都建立了自身的单位矢量场，在重叠区域，需要用平均矢量场作为监督信号。这里需要注意，PAF 是平面矢量场，实现时需要 2 个标量场分别来表示矢量的水平和垂直分量，所以 PAF 分支的输出特征图的通道数为相邻关键点种类数的 2 倍。损失函数是预测 PAF 与真实 PAF 的 L2 范数。

利用关键点热点图和相邻关键点 PAF 进行关键点匹配连接，把属于每个人的关键点组合在一起。关键点热点图只给出所有关键点位置，但并不知道哪些关键点属于同一个人，如何对关键点进行组装形成人体，是难点所在。利用 PAF 能极大地降低问题难度且速度极快，同时很鲁棒，能适应各种复杂场景。以左肘左腕 PAF 为例进行说明，假设图中有 3 个人，左肘和左腕热点图分别检测出每个人的左肘和左腕位置，这些位置对应到左肘左腕 PAF 中。因为每个左肘有三种可能连接方式（连接到所有的左腕），但只有一种连接方式是正确的，利用 PAF 可以量化这些连接方式的正确程度，利用贪婪法很容易选出正确的匹配方式。左肘左腕 PAF 中，任意左肘左腕对之间的矢量场积分值可以量化连接方式的正确程度，积分值越大说明左肘左腕对越有可能属于同一个人。计算出每个左肘左腕对的积分值，首先取最大积 分值作为第 1 个匹配的左肘左腕对；然后将与已经匹配的左肘和左腕相连的其

他左腕或左肘对的积分值置为 0，保证已经匹配的左肘或左腕不再与其他左腕或左肘匹配；接着取最大积分值作为下一个匹配的左肘左腕对，将与本轮匹配的左肘和左腕相连的其他左腕或左肘对的积分值置为 0。一直循环下去，直到所有左肘左腕都匹配。这样就完成了所有人的左肘左腕的匹配。以同样的方式处理所有的 PAF，就完成了人体所有相邻关键点的匹配连接，也就完成了人体关键点组装。也可以采用图论中匈牙利算法进行左肘左腕 PAF 的匹配，读者可以查阅相关资料。实际计算相邻关键点矢量场积分值时采用求和法近似，即对相邻关键点之间的矩形区域的矢量分量求和，和作为积分值。

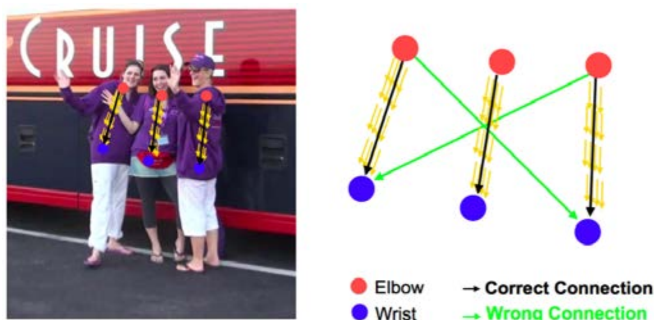


图 12.29 相邻关键点匹配

为了进一步提高性能，也可以采用和单人关键点定位任务中一样的策略，堆叠多个分支网络。具体堆叠方式和沙漏模块有 2 点不同，首先 2 阶段后的分支网络结构和 1 阶段不同，采用 5 个连续的 7×7 卷积代替 3 个连续的 3×3 卷积；其次连接方式采用特征图堆叠方式（Concatenate），而不是逐元素相加（Sum）方式，具体是把上下两个

分支的预测特征图和主干网络特征图 F 堆叠在一起，作为下一阶段的输入。一般采用 2 到 6 个阶段，第 3 个阶段后增益很少。以上介绍的算法就是著名的 OpenPose。

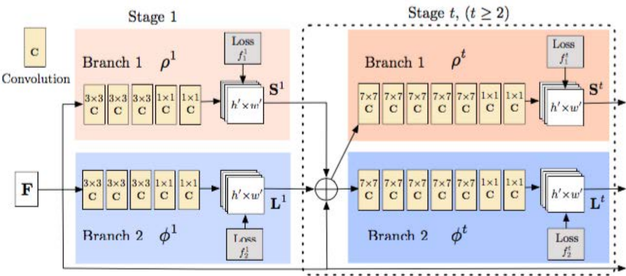


图 12.30 堆叠多个分支网络提高性能

第 13 章

二值化网络

卷积网络取得了巨大成功，广泛应用于图像、语音和语言领域，但它巨大的权重数量（几百兆浮点数）和运算量（上千兆的浮点数乘法）限制了其应用范围。卷积网络主要运行在 GPU 上，很少使用在 CPU 和嵌入式系统（如 ARM）上。在手机上很难运行卷积网络，因此通常采用的做法是，手机先把图像无线传输到云端，云端的 GPU 运行卷积网络，再把结果传回手机。如果卷积网络能在移动设备（如手机）上运行，则会大大促进卷积网络的普及，促进社会发展。为此，我们研发了多种技术来减小卷积网络的权重数量和运算量。第 10 章介绍的轻量网络就是一种具有代表性的技术，通过巧妙的网络结构，直接减小权重数量和运算量。还有一种技术是通过量化权重，来减小权重的存储空间，但并不减小其数量，比如权重是浮点数，需要 32 个比特（bit）；如果变为整数，只需要 8 个 bit；进一步变为 $\{+1, -1\}$，仅需要 1 个 bit，这大大减小了存储空间。卷积网络的主要运算是激活和权重的矩阵乘法（浮点数乘法），如果能把浮点数乘法变为浮点数加减法，甚至是逻辑运算，就能大大提高运算速度、降低功耗。二值化

网络（XNOR Net）是一种量化技术，通过把权重和激活量化为{+1, −1}，减小了存储空间（原来的 1/15）、提高了运算速度（原来的 10 倍）和减小了功耗（原来的 1/30）。

13.1 权重二值化

最简单的二值化网络是只对权重进行二值化，即权重只取+1 或−1。训练网络时，先计算权重的梯度，然后进行权重更新，而为了使训练过程稳定收敛，权重每次的更新量很小。这样一来，如果训练过程中权重只取+1 或−1，由于更新量很小，很难使权重从+1 改变为−1，权重值在训练过程中几乎不会发生变化，不能进行学习。为此，在训练过程中需要引入浮点数权重，利用浮点数权重进行更新。二值化权重是对浮点数权重进行二值化得到的，最简单也最常用的二值化函数是符号函数 sign，即浮点数权重 w_r 大于等于 0 时，二值化权重 w_b 为+1，否则为−1。

$$w_b = sign(w_r) \quad (13.1)$$

网络训练好后，只需保存二值化权重 w_b，浮点数权重 w_r 不需保存。进行预测时，网络前向运算为：

$$\begin{gathered} a_1 = xw_{b1},\ h_1 = f(a_1), \\ \vdots \\ a_i = h_{i-1}w_{bi},\ h_i = f(a_i), \quad (13.2) \\ \vdots \\ a_n = h_{n-1}w_{bn},\ h_n = a_n \end{gathered}$$

其中 x 是输入，h_n 是网络预测值，f(x)是非线性激活函数。预测过程和常规网络一样，只是权重采用二值化权重 w_b。由于权重为{ +1, −1 }，矩阵乘法 $\boldsymbol{a}_i=h_{i-1}w_{bi}$ 变为浮点数加减法，能降低功耗和提高速度。

网络训练时，由于只能对浮点数权重 w_r 进行更新，故前向运算为：

$$
\begin{gathered}
w_{b1}=sign(w_{r1}),\ a_1=xw_{b1},\ h_1=f(a_1), \\
\vdots \\
w_{bi}=sign(w_{ri}),\ a_i=h_{i-1}w_{bi},\ h_i=f(a_i) \\
\vdots \\
w_{bn}=sign(w_{rn}),\ a_n=h_{n-1}w_{bn},\ h_n=a_n \\
c=L(h_n,y_0)
\end{gathered}
\tag{13.3}
$$

y_0 是 x 对应标签，L(h_0, y_0)是损失函数。先对浮点数权重 w_r 进行二值化，得到二值化权重 w_b，然后进行矩阵乘法。难点是计算损失对 w_r 的梯度，根据链式法则得到损失 c 对权重 w_b 的梯度，损失 c 对权重 w_r 的梯度为：

$$
\frac{\partial c}{\partial w_r}=\frac{\partial c}{\partial w_b}\frac{\partial\, sign(w_r)}{\partial w_r} \tag{13.4}
$$

需要计算符号函数的偏导数，由于符号函数的偏导数几乎处处为 0，导致损失 c 对权重 w_r 的梯度也几乎处处为 0，因此权重 w_r 难以更新。为了解决此问题，引入直通估计(Straight-Through Estimator，STE)，把符号函数的偏导数定义为：

$$
\frac{\partial\, sign(w_r)}{\partial w_r}=1_{|w_r|\le 1} \tag{13.5}
$$

当权重 w_r 的绝对值小于等于 1 时，梯度为 1，否则为 0。因此，当权重 w_r 绝对值小于等于 1 时，损失 c 对权重 w_r 的梯度直接等于损失 c 对权重 w_b 的梯度，这就是直通估计。权重 w_r 的绝对值大于 1 时，损失 c 对权重 w_r 的梯度为 0。

训练过程如下：首先权重 w_r 随机初始化；然后采用公式(13.3)进行前向计算，计算损失 c 对权重 w_b 的梯度，根据公式(13.4)和公式(13.5)计算损失 c 对权重 w_r 的梯度，权重 w_r 进行更新，完成一次迭代。

前面是以传统网络为例进行介绍的，卷积网络也完全一样，因为卷积运算经过整理后变为矩阵乘法，前向计算和公式(13.3)一样。由于代码很简单，此处省略，读者可以参考随书代码。代码实现时，没有进行优化，只展示算法原理。

13.2 XNOR 网络

仅对权重进行二值化，矩阵乘法变为浮点数加减法，虽然理论上存储量能获得 32 倍的提升，但提速效果十分有限，不到 2 倍。如果激活二值化为｛＋1，－1｝，则矩阵乘法只需要进行逻辑同或（XNOR）运算和对＋1 进行计数，极大提高运算速度。XNOR 是数字逻辑运算，有 2 个输入端和 1 个输出端，当输入电平相同时，输出为高电平（逻辑 1）。矩阵乘法由向量点积构成，假设 2 个二值化向量为 v1=[＋1,－1，＋1，＋1]和 v2=[－1，－1，＋1，－1]，则点积为(＋1)×(－1)＋(－1)×(－1)＋(＋1)×(＋1)＋(＋1)×(－1)=(－1)＋(＋1)＋(＋1)＋(－1)=2

×num(＋1)－length(v1)=0。＋1 和－1 之间的乘法和 XNOR 一致，同号为＋1，异号为－1。对 XNOR 结果中的＋1 进行计数，则点积为 2 倍的＋1 数目减去向量的长度，由于向量长度是固定值，所以只需记录＋1 数目。加入激活二值化的前向运算为：

$$
\begin{gathered}
w_{b1}=sign(w_{r1}),\ \ a_1=xw_{b1},\ \ h_{b1}=sign(a_1),\\
\vdots\\
w_{bi}=sign(w_{ri}),\ \ a_i=h_{b(i-1)}w_{bi},\ \ h_{bi}=sign(a_i)\\
\vdots\\
w_{bn}=sign(w_{rn}),\ \ a_n=h_{b(n-1)}w_{bn},\ \ h_n=a_n\\
c=L(h_n,y_0)
\end{gathered}
\quad (13.6)
$$

有几个细节值得注意，对激活进行二值化，相当于引入了非线性，称为二值化激活，所以不再需要非线性激活函数。输入 x 不能进行二值化，否则丢失太多信息，效果极差。最后一层得分向量 a_n 也不能进行二值化，否则得分向量不是＋1 就是－1，无法分类。

XNOR 网络在训练过程和权重二值化网络一样，只是非线性激活函数被符号函数取代，利用公式(13.5)进行激活梯度的传递。

13.3　权重尺度化

XNOR 网络极大地减小了存储空间和功耗，并提高了运算速度，但由于二值化带来信息丢失，和浮点数网络相比，性能下降很大。为了提高性能，可以对权重进行尺度化，这个过程仅增加极少的运算量和存储空间。矩阵乘法由点积构成，二值化尺度化后的权重和输入的点积尽可能与浮点数权重和输入的点积接近，这样权重量化带来的信

息丢失就会最小化，网络性能损失最小。假设浮点数权重向量为 w_r，对应的二值化向量为 w_b=sign(w_r)，尺度因子为 α（1 个正浮点数），输入向量为 x_r，则量化后的点积 $pb=\alpha x_r^{T}w_b$ 要和原始点积 $pr=x_r^{T}w_r$ 尽量接近，即要求向量差 $pr-pb$ 的 L2 范数最小，通过不太复杂的数学运算，得最优 α：

$$\alpha=\frac{1}{n}\sum|\boldsymbol{w}_{ri}| \quad (13.7)$$

即向量 w_r 绝对值的平均值。

公式(13.7)是权重为向量时的结论，而前向运算中权重是矩阵，此时只需把权重矩阵看成是列向量的集合（线性代数中经常这样），每个列向量计算尺度因子 α_i，则权重矩阵的尺度因子是各个尺度因子 α_i 组成的行向量 $\boldsymbol{a}$。前向运算为：

$$\begin{gathered}
w_{b1}=sign(w_{r1}),\ \alpha_1=scale(w_{r1}),\ a_1=\alpha_1 x w_{b1},\ h_{b1}=sign(a_1), \\
\vdots \\
w_{bi}=sign(w_{ri}),\ \alpha_i=scale(w_{ri}),\ a_i=\alpha_i h_{b(i-1)} w_{bi},\ h_{bi}=sign(a_i) \\
\vdots \\
w_{bn}=sign(w_{rn}),\ \alpha_n=scale(w_{rn}),\ a_n=\alpha_n h_{b(n-1)} w_{bn},\ h_n=a_n \\
c=L(h_n,y_0)
\end{gathered} \quad (13.8)$$

即先计算二值化权重矩阵和权重尺度向量，然后矩阵乘法，最后二值化激活。矩阵乘法是先计算 2 个二值矩阵的乘积（XNOR 运算），然后再与权重尺度向量进行逐元素的乘法，由于权重尺度是浮点数，这会增加一定的运算量，增加的乘法运算量等于矩阵元素数量，与矩阵乘法运算量相比，占比小，所以影响也较小。

进行梯度反向传播时有个技巧，即把二值化权重矩阵和尺度因子向量的逐元素乘积矩阵 w_α作为一个整体进行计算，公式如下：

$$
\begin{gathered}
w_{\alpha 1}=signscale(w_{r1}),\ \ a_1=xw_{\alpha 1},\ \ h_{b1}=sign(a_1),\\
\vdots\\
w_{\alpha i}=signscale(w_{ri}),\ \ a_i=h_{b(i-1)}w_{\alpha i},\ \ h_{bi}=sign(a_i)\\
\vdots\\
w_{\alpha n}=signscale(w_{rn}),\ \ a_n=h_{b(n-1)}w_{\alpha n},\ \ b_{bn}=a_n\\
c=L(h_n,y_0)
\end{gathered}
\tag{13.9}
$$

根据链式法则得到损失 c 对权重 w_α的梯度，损失 c 对权重 w_r 的梯度为：

$$
\frac{\partial c}{\partial w_r}=\frac{\partial c}{\partial w_\alpha}\frac{\partial\, signscale(w_r)}{\partial w_r}=\frac{\partial c}{\partial w_\alpha}[\frac{1}{n}+\alpha\frac{\partial\, sign(w_r)}{\partial w_r}] \tag{13.10}
$$

训练过程和权重二值化网络完全一样。

为了进一步提高性能，对激活进行二值化时可以引入尺度因子。这个技巧对性能提升比较小，但引入的运算量和临时存储空间比较大，读者可根据情况进行选择。

实际应用时，由于网络第 1 层和最后 1 层的权重数量少，一般不进行权重量化，而是采用浮点数权重，性能获得极大提升。

量化技术应用于卷积网络时，有几点需要注意。卷积网络模块一般包括批量归一化层 B、二值化激活层 A、卷积层 C 和池化层 P，这些层的顺序对性能影响很大，浮点数网络的顺序为 C、B、A、P，但对于量化网络，顺序应为 B、A、C、P。因为经过二值化激活层 A 后，

激活不是＋1 就是－1，如果紧跟池化层 P，则激活几乎都是＋1。卷积网络中大量存在 1×1 卷积，但量化网络中，如果采用 1×1 卷积，对性能损失很大，所以量化网络中不能有 1×1 卷积。由于这个原因，量化网络很难对轻量网络进行加速。二值化激活层可以取代非线性激活层，但对复杂网络（如 AlexNet、VGG 和 ResNet），在卷积层后插入非线性层（ReLU）能提高性能。



微信连接



回复“深度学习”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书

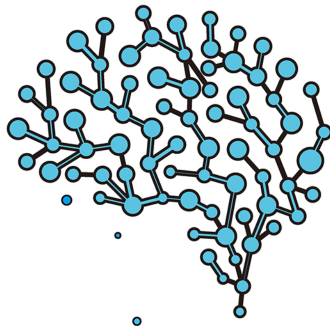


QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈



卷积神经网络的Python实现

卷积神经网络是深度学习中的重要模型，掀起了此次人工智能的浪潮，广泛应用于图像、语音、翻译和强化学习领域。

本书用极少的数学知识，深入浅出地介绍了机器学习、卷积神经网络的相关概念以及实践中特别重要的数据预处理。书中没有借助深度学习库，完全使用Python语言基于NumPy库实现了神经网络和卷积神经网络，并给出了全部代码。为了方便读者理解深度学习和更好地使用深度学习库，如TensorFlow，书中特别对误差反向传播算法和神经网络的优化方法进行了深入分析。在此基础上，本书进一步实现了经典的VGG网络和移动端MobileNetV2网络，同时介绍了GoogLeNet、ResNet和SENet。

图灵社区：iTuring.cn
微　　博：@图灵教育　@图灵社区

分类建议	计算机 / 人工智能 / CNN

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-49756-7
定价：49.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks